SMIILE: Smart Module Integration for IoT Programming Logic and Environment

Samuel Sungmin Cho*, Tami Farber*, Clinton Schultz*, Nicholas Caporusso*, Myoungkyu Song†
*Northern Kentucky University, †University of Nebraska at Omaha
Email: *chos5@nku.edu, *farbert1@nku.edu, *schultzc10@nku.edu, *caporusson1@nku.edu, †myoungkyu@unomaha.edu

Abstract—**Internet of Things (IoT) aims to build a smart environment by using various smart devices, including servers, mobile devices, and sensors to enhance people's life. Understanding how to use different hardware and software components—programming languages, libraries, tools, and configuration setups—is required to build IoT applications due to the heterogeneity of IoT devices. As a result, building IoT applications can add unnecessary burdens to developers because they need to focus not only on the application logic that meets clients' requirements but also on the details of implementing and configuring the services. In this paper, we propose SMIILE, Smart Module Integration for IoT Logic and Environment, to address this issue. In this programming model, IoT developers use the SMIILE language to *describe* their application logic and *prescribe* the programming configuration. Then the SMIILE compiler analyzes the description and prescription to generate deployable configuration and source code files and the documentation necessary to guide IoT programmers in building the applications effectively. In our deployment model, IoT programmers can use SMIILE to supplement their application development process by customizing the libraries to meet their needs or incorporating the SMIILE generated code into their programming environment.**

I. Introduction

Internet of Things (IoT) is about connecting intelligent devices with various capabilities to provide services in many application domains, including logistics, healthcare, transportation, or environmental monitoring [1]. Depending on the application domain, IoT programmers use different type of hardware, for example, servers that normally are equipped with large memory and computing capability, mobile devices that are battery-powered but still have enough computing power to process various tasks, and small sensor devices that are limited in energy, memory, and computing capability [2], [3].

To use different IoT devices, IoT application developers need to use different methods or approaches in programming [4]. IoT application developers mainly use high-level programming languages, such as Python or Java, on integrated development environments (IDE) for intelligent devices with large memory and computing capabilities, but sometimes they have to use low-level programming languages, such as Assembly or C, using simple programming editors for devices with limited memory or computing power. Therefore, for IoT application development, programmers are burdened with implementing the same application logic differently depending on the application domains, hardware choices, programming languages, or programming environments.

Configuration and setup is another burden for IoT programmers; they need to spend their time and efforts on setting up programming configurations for each device. For example, when programmers need to use different communication interfaces, they should use different configuration setups. Devices that use Bluetooth for wireless point to point communication will have dramatically different configuration setup compared with RS-232 or IEEE 802.11 based connection.

We present SMIILE (Smart Module Integration for IoT Programming Logic and Environment). SMIILE tries to address the programmers' burden from the inherent variety of IoT devices and programming methods. SMIILE is an IoT programming model that allows programmers to solely focus on their programming logic by delegating the implementation, environment configuration, and deployment to a set of integrated tools. As an example, when an IoT programmer builds a sensor information acquisition system with a sensor device using a Wireless Internet interface, the programmer should build a detailed application code for the sensor application, generally in C for small memory devices and Python for large memory devices. For the communication setup of Wireless Internet interface, the programmers should understand the hardware's connection to the Internet hardware, and they sometimes need to make a setup file for the device configuration. When the IoT system requires a new communication interface such as Bluetooth, the programmer has no choice but to rewrite all of the configuration code, and if necessary, some of the application code to meet the new requirement.

SMIILE uses a streamlined process of generating deployable source code, configuration code, and if necessary, document files to guide programmers with necessary information about the code, configuration, and hardware system from users' inputs such as code, configuration, and setup files. SMIILE uses a setup file to specify what programming language or hardware environment to use for implementing the application goal. For example, programmers can specify they use C on the Arduino board for sensor devices, or Python on MircoPython for the Internet access, using the SMIILE programming language, describing their requirements in the setup file. As SMIILE system already has code and configuration libraries in the form of a searchable database, SMIILE compiler generates deployable code in Python or C code that uses the sensor or Internet libraries.

The SMIILE code has three sections: prescription, instantiation, and description. In the prescription section, programmers prescribe the programming environment-related information such as type of hardware or programming language. In the instantiation section, programmers instantiate the entities, such as communication interfaces or sensors, that will be used to implement application logic. The description section is used to describe the application logic in the SMIILE programming components. The configuration information and setup files are used to supplement the prescription part of the SMIILE code by allowing programmers to specify the default values or their preferences in generating deployable source code.

Our contributions in this paper are as follows:

- We present the SMIILE programming model and the SMIILE compiler to aid IoT application developers. We explain the benefits of using the SMIILE programming language to generate deployable code, configuration, and documentation from users' prescription and description.
- We present the SMIILE deployment model that can integrate automatically generated or composed from user-configurable SMIILE libraries and users' code.
- We build three examples, I/O, sensor, and Web, to show how SMIILE can reduce IoT programmers' burden by helping to generate deployable code, configuration files, and documents.

The rest of this paper is organized as follows: we describe related work in Section II. We explain the concepts of the SMIILE programming model, the SMIILE processes, and the SMIILE deployment model in III. In Section IV, we show the experimental results by comparing the automatically generated code, configuration, and information files with manually generated ones. Section V concludes this paper.

II. Related Work

The programming model is an abstraction of a software layer to provide data structures and algorithms that enable program expressions [5]. SMIILE is based on this programming model idea to develop a compiler, libraries, and database system to enhance interoperability programming among IoT devices. In our previous research [4], we explored the programming language and its related tools based on a programming model to address the programming burden of IoT programmers.

Domain-Specific Languages (DSL) are used to leverage heterogeneous environment of the IoT platform by abstracting application domains and generating concrete code for deployment. Abstracting the application generation problem of an IoT platform was proposed by García et al [6]. In their research, they showed that they could build a DSL to create interconnected IoT applications among heterogeneous objects such as sensors, mobile devices. The DSL approach helps a user to interconnect many objects without any programming knowledge from the support of the abstraction of special programming language, given the problem domain and its properties are provided. Salihbegovic et al. [7] also present a special language that can help to develop IoT applications using complex protocols, communication, and operating systems. The DSL is used to describe visual notations for IoT applications, and it is an abstraction for formal representation as a meta-model. From our previous research of Chitchat Information Sharing Language (CISL) [8], we presented a DSL language for sharing context information. CISL is also based on a context sharing programming model among IoT devices. SMIILE uses similar concepts such as programming model and using a DSL, but the focus is more on a system description and prescription, code generation, and deployment so that IoT application developers can use without knowing information representation.

One of the main focuses of IoT programming is to enhance interoperability among IoT devices [9]. IoT middleware [10], [11] is mainly used to address some of the interoperability issues. In our research, however, we focus on the more practical aspect of IoT interoperability issues by providing the SMIILE compiler and related tools that IoT programmers can easily use to focus on their application logic. Using standard data exchange formats, such as XML or JSON, is proposed [12]. We use JSON as a format to describe setup and constraints information.

III. SMIILE Programming Model and Process

SMIILE programming model is to use three sections, prescription, instantiation, and description, to specify a programmer's application logic using the SMIILE programming language. The SMIILE compiler analyzes the specifications and generating deployable code by retrieving necessary information from libraries. When the programmer does not specify detailed information about the system, such as port number, pin number, of programming logic, the SMIILE can use configuration or setup to generate deployable code. In doing so, all the necessary configuration code or documentation files are generated.

Listing 1 shows an example of SMIILE code that describes the application logic that senses data from a temperature sensor and transmits the data to NodeMCU hardware through the I2C channel.

```
// Prescription
#platform NodeMCU
#use temperature_sensor
#use I2C

// Instantiation
t = TemperatureSensor()
c = I2C()

// Description
t1 = t.sense()
r = (t1 * 9/5) + 32 // Celsius to Fahrenheit
c.send(r)
```

Listing 1: SMIILE description and prescription

This code has three sections in it. The first section prescribes hardware-related information, including the hardware platform (NodeMCU), the type of sensor that it uses (temperature sensor), and the communication channel (I2C). When any of the prescriptions is missing, a default value will be used. Programmers can set these values using configuration files. The second section, the instantiation section, describes what entities, i.e., objects, will be used, and the variables to reference the entities. In this example, reference *t* and *c* are used to specify the usage of temperature sensor and I2C communication interface respectively. The final section describes the application logic that converts the temperature data in Celsius degree to Fahrenheit. In this section, references in the instantiation section or any temporary values can be used to describe the control logic of the application.

```
{
  language = C
  output = LCD
  temperature_sensor = MCP9808
}
```

Listing 2: SMIILE configuration

Listing 2 shows the configuration for the SMIILE code. In this configuration file example, we specify that the SMIILE compiler should generate deployable code in the C programming language. Also, we can specify the output and the sensor hardware that we will use when programmers do not specify them: the result will be displayed using an LCD with the 'output' parameter, and the sensor data will be acquired from an MCP9808 temperature sensor.

```
{
  programmer = Samuel Cho
  library_location = /user/lib
  library_name = smiile.db
}
```

Listing 3: SMIILE setup

SMIILE needs the setup file that SMIILE uses to get any global information. Listing 3 shows the setup file that contains the name of a programmer, the location of a library, and the name of the library.

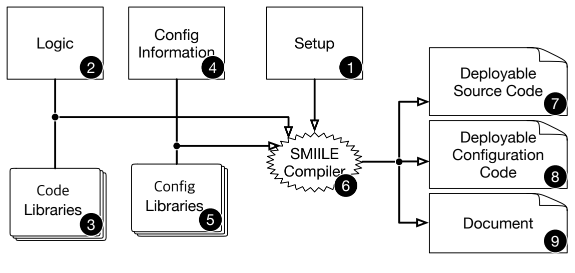

Fig. 1: SMIILE Process

This programming model allows SMIILE to handle interoperability among IoT devices effectively, as programmers prescribe the hardware requirements to build IoT applications and describe the detailed application logic, both using the high-level SMIILE programming language.

Fig 1 shows the SMIILE process. programmers write setup file ❶ that contains all the setup related information. They describe the application logic using SMIILE language ❷ and configuration information ❹. The configuration information and application logic is analyzed by the SMIILE compiler ❻ to incorporate related code libraries ❸ and configuration libraries ❺ to generate deployable code ❼, configuration code ❽, and documentation ❾.

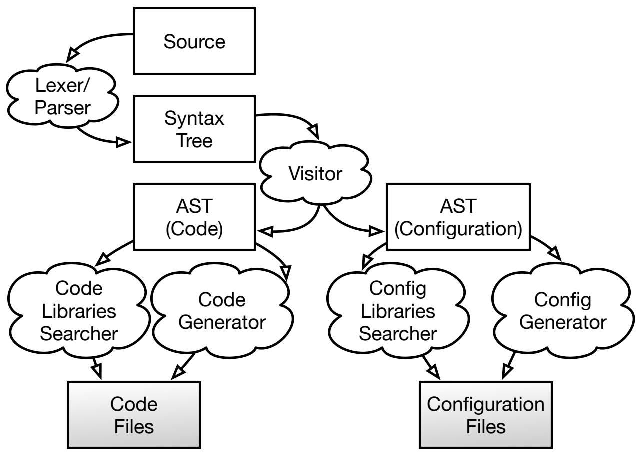

Fig. 2: SMIILE Compiler Process

Fig 2 shows how the SMIILE compiler generates deployable files from input files in more detail. The input source is analyzed using a lexer/parser to get a syntax tree. We use a visitor design pattern for generating ASTs; A visitor visits each tree to generate both a code AST (abstract syntax tree) and a configuration AST. When the code AST has nodes that use an API (application programming interface) to implement the functionality, or the SMIILE compiler needs to find the code that requires API functions, the SMIILE compiler finds the matched code in code libraries ❸ in Fig 1. Likewise, the configuration files are analyzed, and related information is retrieved from the config libraries ❺ in Fig 1 by the SMIILE compiler.

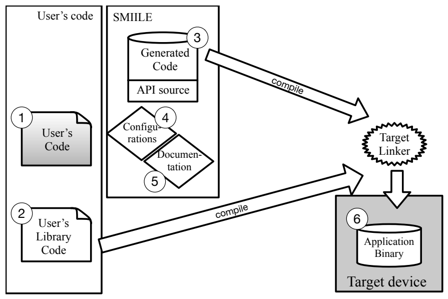

Fig. 3: SMIILE Deployment Process

The generated files should be deployed and installed on the target hardware. However, if necessary, programmers can integrate the SMIILE process with their own programming process. Fig 3 shows this deployment model. In this model, programmers can use their code ① or other's library code ② with generated SMIILE code files ③ easily; they are separately compiled and then linked using configuration files ④. The automatically generated documentation files can help programmers to understand and facilitate the whole installation process ⑤. Users' code ① and library files ② in Fig 3 are sometimes necessary because the code generated from

SMIILE may have limitations. The limitation is mainly from the fact that SMIILE has an API based structure.

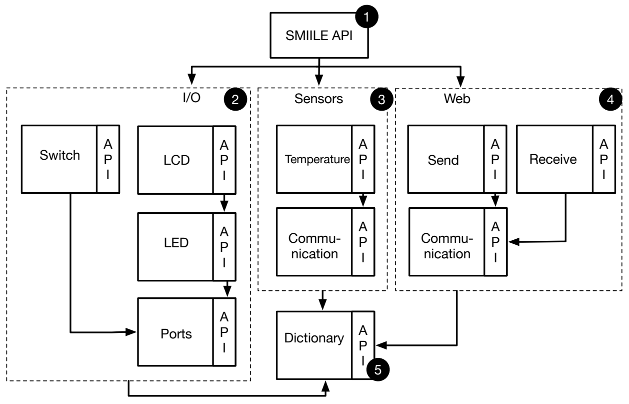

Fig. 4: SMIILE API based structure

Fig. 4 shows how the objects are structured and organized in the libraries. The example in Listing 1 uses a *TemperatureSensor* object that has *sense()* API. SMIILE has various modules that have multiple objects and APIs such as I/O, Sensors, and Web. This module-based SMIILE library structure makes the SMIILE tools easy to extend features because new features can be made available with new SMIILE APIs to the application developers who request them. The APIs implementation feature also facilitates the portability, because the differences between platforms or hardware architectures are hidden behind the APIs.

In the SMIILE programming model, programmers use APIs in modules for describing their application logic, and the SMIILE compiler generates code that uses with these module APIs. The compiler also adds the body of the code in the generated source file. The benefit of API based application development is that it abstracts all the interactions among modules into a simple set of functions for application developers. Each SMIILE API function's signature is defined as they are the interfaces that developers use for their applications. Each SMIILE API function call does not change global states. The only exception to this rule occurs during error processing. The SMIILE programming model cannot use exceptions or another language-specific error processing mechanism. There is one global state that describes the error; when errors occur, developers can read the state to identify the type of error.

In the SMIILE programming model, we see a library as an abstraction of a module, which makes the dependency among modules correspond to the dependencies among libraries. This approach hides the differences in various programming languages that use different names for supporting modular programming. For example, the C++ programming language uses the name 'object file' as the result of compilation from a compilation unit and 'library' for a collection of object files while the Java programming language uses the name 'class file' and 'jar' to represent similar concepts. In the SMIILE model, we define a library as the compilation of a module. A module can have multiple object or class files, but a module should export only a single set of APIs.

Using this structure, SMIILE programming model benefits from the controlled dependency that emphasizes the rules 1) the APIs (interface) and the implementations are separated, and 2) the modules maintain simple dependency relationships that are controllable by application developers. The modules programmers should access are properly defined and export only the objects and APIs with well-defined behavior. The separation of APIs (interface) layer and implementation layer in the SMIILE model gives the freedom of modifications in library code without changing the user's code. It also gives freedom of implementation differences on various hardware or software platforms.

IV. Experiments

In this section, we use three different examples to show how the SMIILE programming model and the SMIILE process can be applied to generate application code, configuration files, and documentation files. The first example is controlling I/O systems. In this example, we use the SMIILE programming model components (prescriptions, instantiation, and descriptions) to turn on or off LED lights by controlling a switch. The second example is detecting environmental information through a sensor. We use a temperature sensor to read the sensor data. The last example is sending information to a web server using a web interface.

The effectiveness or performance of the SMIILE programming model is assessed by measuring the total lines of automatically generated code and comparing it with the lines of the source code in the SMIILE language. This will show how the SMIILE system can reduce the coding and documenting burden of programmers.

A. I/O - LED and Switch

In the first example (I/O), an IoT application developer selects an Arduino board [13], to make an application that turns on or off LED lights by clicking a switch. The developer can easily change the prescriptions section in the SMIILE source code to generate the deployable code that can be executed on different hardware such as MicroPython [14] or ESP8266 [15].

In the code shown in Listing 4, the LED (led_armed) is turned on or off when a user clicks a button. The prescription part has three lines of code to show programmers' hardware usage. The programmer uses an Arduino hardware (line 1), an LED (line 2), and a Button (line 3). The instantiation section (lines 5 - 7) identifies the three entities, an LED, Button, and one variable that stores the armed state. The logic section (lines 9 - 16) describes the application logic using SMIILE programming components such as selections (if/else), loops (while), assignments (=), or comparisons (==). Any code after "//" will be ignored.

Listing 5 shows an example configuration file for Arduino, including the project file or related files and pin assignments. The configuration file that SMIILE generates can contain the information from the setup file for validation purposes.

```
// Prescriptions
#platform Arduino
#use LED
#use Button

// Instantiation
led_armed = LED()
button = Button()
arm_state = False

// Descriptions
while True:
 if button.value == 1 {
     if arm_state == False {
         led_armed.on()
         arm_state = True
     }
     else {
         led_armed.off()
         arm_state = False
     }
 }
 wait(1)
```

Listing 4: I/O example

If necessary, programmers manually build and start Arduino projects, and they can include the generated source file.

```
/**********************
IO_example.ino
Samuel Cho
*/

#define VERSION_NUMBER 1
#define SENSOR_PIN 14
```

Listing 5: I/O configuration example

Notice that when programmers do not specify the port number to control the LED or Button, as in the Listing 2, the SMIILE system assigns the port number as in the Listing 5. The SMIILE system also stores any information that is needed to build the application—installation instructions, project information, hardware configuration information, or any information stored in the database system—in a documentation file. Listing 6 shows an example of generated documentation in a Markdown file format [16]. We used three dots (line 10) to notify additional lines are removed.

```
1 * This is automatically generated from SMIILE system.
2 * Do not edit this file, re-generate this file instead.
3
4 ## I/O (LED and Button)
5
6 # Instruction
7 The input pin should be connected to Arduino pin 32
8 The output pin should be connected to Arduino pin 33
9
10 ...
11
12 # Reference
13 Refer to https://forum.arduino.cc/topic=410 for your
       reference.
```

Listing 6: I/O document example

The setup file has all the necessary information to make an IoT project, including the project name and programmer name. The setup file also contains the default values when the programmers of SMIILE system specify any hardware related configurations such as pin numbers. Listing 7 is an example of the setup file.

```
project_name = I/O (LED and Button)
programmer = Samuel Cho
version = 0.1

# Arudino
default_input = 32
default_output = 33
```

Listing 7: I/O setup example

B. Sensors - Temperature and Motion

IoT programs sense environmental situation to analyze it and take actions depending on the analysis results. Therefore, the processing of sensor information is a critical part of IoT programming. The SMIILE programming model provides the features of sensing and processing the sensor information automatically or manually. Listing 8 shows an example of sensing temperature information. The code alerts users by sending an email when the sensed temperature is above a certain value. Based on the prescriptions (line 1 – line 3), the SMIILE decides the hardware platform and the temperatures sensor type to use, and the SMIILE can use the email information from the setup file. The SMIILE automatically generates the detailed pin configuration, email senders and receivers, and any related information in the documentation file so that programmers can verify.

```
1 // Prescriptions
2 #platform NodeMCU
3 #use temperature_sensor
4 #use email
5
6 // Instantiation
7 t = TemperatureSensor()
8 e = email()
9
10 // Descriptions
11 while True {
12   t1 = t.sense()
13   if t1 > 50 {
14     e.send("Temperature too hot")
15   }
16   sleep(10)
17 }
```

Listing 8: SMIILE sensor example

C. Web application

Listing 9 shows an example when programmers use a web interface to send the temperature information to a web server. The prescription section specifies the hardware platform (NodeMCU), usage of a temperature sensor, and Web as a communication interface. The configuration details, such as the web API usage, need to be described in the setup file. However, when programmers do not specify all the necessary information, the implementation details are synthesized from the code database libraries to help programmers understand how the system is configured and worked.

Table I shows the results when programmers use the SMIILE system to generate code, configuration, and documentation using the SMIILE programming model. For the first example (I/O), the SMIILE programming code count is 16, but the SMIILE compiler generates 32 lines of code to make the code reduction rate of 50%. The configuration file is also

```
// Prescriptions
#platform NodeMCU
#use temperature_sensor
#use web

// Instantiation
t = TemperatureSensor()
e = web("http://example.com")

// Descriptions
while True {
  t1 = t.sense()
  e.send(t1)
  sleep(10)
}
```

Listing 9: SMIILE web example

generated, but programmers should make the configuration file to control the hardware boards. This makes it hard for us to define the reduction rate; so we specified them as 0.0 reduction rate. The documentation is automatically generated from the SMIILE system, so we specified it as non available (N/A).

The I/O and sensor examples show good size reduction. However, the Web examples shows somewhat dramatic code size reduction, as web programming requires various and sometimes lengthy boilerplate code. The SMIILE system automatically generates most of the boilerplate code so that programmers can focus only on writing application logic implementation in the SMIILE programming language.

TABLE I: Reduction rate (%) in code

	Code		Configuration		Documentation	
	Size	Rate (%)	Size	Rate (%)	Size	Rate (%)
I/O	16	19.2	28	0.0	121	N/A
Sensor	17	22.1	26	0.0	192	N/A
Web	16	72.5	75	0.0	241	N/A

V. Conclusion and Future Work

The Internet of Things (IoT) programming requires programmers to understand various hardware specifications, capabilities, and configuration setups. It also requires different programming tools, including programming languages and libraries. SMIILE can alleviate this programming and configuration burden by allowing IoT programmers to use SMIILE language to prescribe the programming environment and describe application logic to effectively and quickly generate code, configuration, and documentation for the IoT application.

In this work, we present a SMIILE that allows IoT application developers to focus solely on the tasks at hand while delegating most of the software and hardware related configuration, setup, or detailed implementation to the SMIILE tools and libraries. We propose the SMIILE process that automatically generates code files, configurations, and documentations from users' prescriptions of their intended system and description of their application logic. We also proposed the SMIILE deployment process to combine the users' code files or configurations with SMIILE generated files. While default SMIILE system libraries or setup system files can be used to build IoT application software, users can extend, modify, or revise the existing components to fit their needs. In our experiments, we showed three examples, I/O, sensors, and web applications, to demonstrate that the SMIILE system could reduce programmers coding burden up to 72.5% by writing only program logic and configurations, which are processed by the SMIILE compiler and library processing system. To improve upon our work, we plan to add more libraries to support diverse CPUs, hardware boards, sensors, and I/O systems. We also plan to make the system generate a more detailed documentation system so that programmers can be effectively guided to build IoT systems to meet their application requirements.

References

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey." *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] S. Cho and C. Julien, "Size Efficient Big Data Sharing Among Internet of Things Devices," in *PerCom Workshop BICA*, Jan. 2017, pp. 1–6.

[3] ——, "CHITCHAT - Navigating tradeoffs in device-to-device context sharing." in *PerCom'16: Proceedings of the 2016 IEEE International Conference on Pervasive Computing and Communications*. IEEE, 2016, pp. 1–10.

[4] S. Cho, "Navigating Tradeoffs in Context Sharing Among the Internet of Things," Ph.D. dissertation, The University of Texas at Austin, The University of Texas at Austin, Aug. 2016.

[5] P. McCormick, R. Barrett, and B. de Supinski, "Programming Models," Los Alamos National Laboratory, Tech. Rep. 474731, Mar. 2011.

[6] C. G. García, B. C. P. G-Bustelo, J. P. Espada, and G. Cueva-Fernandez, "Midgar: Generation of heterogeneous objects interconnecting applications. a domain specific language proposal for internet of things scenarios," *Computer Networks*, vol. 64, pp. 143–158, 2014.

[7] A. Salihbegovic, T. Eterovic, E. Kaljic, and S. Ribic, "Design of a domain specific language and ide for internet of things applications," in *MIPRO 2015*.

[8] S. S. Cho, M. Song, and C. Julien, "Intelligent Information Sharing Among IoT Devices: Models, Strategies, and Language," ser. 18th IEEE EIT Conference.

[9] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2347 – 2376, 2015.

[10] C. Perera, P. P. Jayaraman, A. Zaslavsky, D. Georgakopoulos, and P. Christen, "MOSDEN: An Internet of Things Middleware for Resource Constrained Mobile Devices," ser. HICSS'14: Proceedings of the 47th Annual Hawaii International Conference on System Sciences, 2014, pp. 1053 – 1062.

[11] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 3, no. 1, pp. 70 – 95, 2016.

[12] N. Nurseitov, M. Paulson, R. Reynolds, and c. izurieta, "Comparison of JSON and XML Data Interchange Formats - A Case Study." ser. CAINE, 2009.

[13] D. Kushner, "The making of arduino," *IEEE spectrum*, vol. 26, 2011.

[14] M. Khamphroo, N. Kwankeo, K. Kaemarungsi, and K. Fukawa, "Micropython-based educational mobile robot for computer coding learning," in *2017 8th International Conference of Information and Communication Technology for Embedded Systems (IC-ICTES)*. IEEE, 2017, pp. 1–6.

[15] M. Schwartz, *Internet of Things with ESP8266*. Packt Publishing Ltd, 2016.

[16] J. Voegler, J. Bornschein, and G. Weber, "Markdown–a simple syntax for transcription of accessible study materials," in *International Conference on Computers for Handicapped Persons*. Springer, 2014, pp. 545–548.