An Automatic Approach for Transforming IoT Applications to RESTful Services on the Cloud

Yu Zhao[1], Ying Zou[1], Joanna Ng[2], and Daniel Alencar da Costa[1]

[1]Queen's University, Kingston, Canada
yu.zhao@queensu.ca,ying.zou@queensu.ca,daniel.alencar@queensu.ca
[2]CAS Research, IBM Canada Software Laboratory, Markham, Canada
jwng@ca.ibm.com

Abstract. Internet of Things (IoT) devices are prevalent in all aspects of our lives, *e.g.*, thermostat and smart lights. Nowadays, IoT devices are controlled by various end-user applications. There is a lack of a standard interface that allows the communication among various IoT devices. In this context, the functionalities of IoT devices may be published as IoT services. IoT services are RESTful services that connect to IoT devices. The uniform interface of IoT services allows them to be integrated with existing applications. We propose an approach that automatically transforms functionalities of IoT devices to IoT services hosted on the cloud. Our approach identifies the code methods from IoT applications that have to be transformed and also extracts service specifications (*e.g.*, input / output parameters) from these methods. Our case study result shows that our approach obtains a precision and a recall above 70%. The identified methods and service specifications are converted to IoT services. Our approach generates IoT services with an accuracy of 96%.

Keywords: IoT, RESTful services, cloud platform, code analysis

1 Introduction

The inter-connected physical devices, *i.e.*, the Internet of Things (IoT) devices, are prevalent in several aspects of our lives. For example, IoT devices may sense nearby environments (*e.g.*, obtain the temperature) and react upon an end-user's request to change the physical environment (*e.g.*, turn on the light). IoT applications are designed by application developers to provide functionalities in IoT devices, *e.g.*, to sense the temperature. In the meanwhile, the Internet has turned into a global infrastructure to host heterogeneous web services. End-users may use web services to perform various on-line activities, such as on-line shopping and banking. With web services and IoT devices combined, the possibilities to ease our daily lives increase in magnitude. For instance, an on-line grocery order can be made based on a food consumption alert that is triggered by analyzing the data read from a fridge sensor. However, this combination is not without its limitations. For example, end-users must install a large number of proprietary end-user applications (*e.g.*, mobile applications) on smart phones or computers to access the information of IoT applications in IoT devices. In addition, the diverse end-user applications lack a standard interface to allow the communication among various IoT devices and web services. Therefore, it is not trivial to integrate IoT devices with existing applications [15].

To ease the integration of IoT applications, we are interested in transforming IoT applications to IoT services, using the service-oriented architecture (SOA) to provide the functionalities offered by IoT devices. In particular, SOA based IoT services have two main advantages: (1) interoperability, which allows IoT services to exchange information with web services using a structured data format; (2) easy integration with existing applications due to the uniform interface of IoT services. Research effort has been invested on approaches to provide IoT services for end-users [4][9][15]. Nonetheless, most of these approaches run the IoT services on the IoT devices [4][9][15], which are not optimal, since IoT devices are typically designed with limited resources, *e.g.*, low battery capacity and processing power [15]. In addition, the complexity of SOA standards (*e.g.*, the verbose data format) generates energy and latency overheads in IoT devices that lead developers to spend extra effort when designing IoT services.

To overcome these practical limitations, we focus on automatically transforming the functionalities of IoT devices to IoT services. IoT services are designed using the RESTful paradigm. We use the cloud platform to host IoT services. In contrast to the resource limited IoT devices, the cloud platform has massive storage, high speed network and huge computing power. Furthermore, the cloud platform has the potential to host numerous IoT services and connect IoT devices as well as processing IoT data [16]. Additionally, the functionalities of IoT devices may be managed by standard APIs over the cloud, which may be accessed by end-users from any place.

More specifically, we analyze the source code of IoT applications to identify methods that can be controlled or accessed by end-users. Our approach further extracts the service specifications of the corresponding IoT services. A service specification describes the interface of an IoT service and is composed of three parts: service name, HTTP function and input (or output) parameters. To allow developers to modify the generated service specifications, we also propose a service schema that describes the service specifications of IoT services. The service schema identifies which data of IoT devices that should be stored in the cloud. Moreover, we use the service schema to instantiate IoT services with friendly user interfaces.

We evaluate the effectiveness of our approach through two case studies. Our results reveal that we can identify code methods that should be transformed with a precision of 75% and a recall of 72%. We can also extract service specifications from the source code of IoT applications with a precision of 82% and a recall of 81%. Our approach generates IoT services from IoT applications with an accuracy of 96%. These results show that our approach can accurately transform IoT device functionalities to IoT services.

Paper Organization. In Section 2, we present the background of the paper. In Section 3, we give an overview of our approach to generate IoT services. In

Section 4, we describe our case studies. We summarize the related research in Section 5. Finally, we conclude our work in Section 6.

2 Background

In this section, we provide background material about IoT devices, web services, the programming structure of the source code of IoT applications and IoT services.

2.1 IoT Devices

An IoT device is a physical item that is embedded with a computing system and can be controlled remotely through Bluetooth or Wi-Fi. In our approach, we consider three classes of IoT devices: sensors, actuators and composite devices [11]. A sensor can measure the physical properties of a physical environment at a constant frequency, while an actuator is an IoT device that is controlled by end-users who may change some of its physical properties. For example, a sensor can sense the temperature, while an actuator can receive a command to turn on the light. Finally, a composite device is composed of both sensors and actuators. For example, a thermostat is an IoT device which senses the temperature and may be requested to change the temperature.

2.2 Web Services

A web service is a software component that allows machine-to-machine communication through the world wide web. This communication may be implemented using the Representational State Transfer (REST) [17] architecture style. RESTful services typically use HTTP as the underlying protocol to transfer resources. A resource is located by a Universal Resource Locator (URL). Resources may have various representations, *e.g.*, JSON and XML. To use the resources that are available in the web, clients (*i.e.*, applications) send requests using HTTP functions. The available HTTP functions are GET, POST, PUT and DELETE. The GET function requests a read only access to a resource, while the POST function is used to create a new resource. The PUT function is used to update an existing resource, while the DELETE function is used to remove a resource.

2.3 Programming Structure of the Source Code of IoT Applications

The methods in the source code of IoT applications can be classified into two types: *internal methods* and *external methods*. An *internal method* is related to the set up of an IoT device and is only consumed by methods within the IoT device (*e.g.*, an *init* method to set up an IoT device). An *external method* works as an IoT device interface that can communicate with the cloud. The input variables of an external method may represent the input commands of an actuator (*e.g.*, to turn on the light), while the returned variables may represent the sensed data of a sensor (*e.g.*, the sensed temperature). Since an external method allows end-users to control an IoT device or obtain information from an IoT device, it is possible to transform such a method to an IoT service.

```
def led_control(status):
    if status == "ON":
        turn_led_on()
    elif status == "OFF":
        turn_led_off()
```

(a) External method 1

```
def getTemperature():
    temp ← methods to get
              temperature
    return temp
```

(b) External method 2

Fig. 1: Examples of external methods

Figure 1 shows examples of external methods that are extracted from the hackster.io website.[1] Hackster.io is a website that shares projects on embedded devices (*e.g.*, Raspberry Pi). In Figure 1, the names of the methods describe the methods' intent (*i.e.*, `led_control` and `getTemperature`). The `led_control` method[2] (Figure 1a) can receive commands from the cloud (*i.e.*, by using the `status` variable). This method uses an *if-else statement* to identify whether the led has to be turned on or off depending on the `status` variable. The `getTemperature` method[3] (Figure 1b) retrieves the temperature from a sensor. A developer can define methods within the external method to send the sensed temperature values to the cloud, *e.g.*, `send(temperature, url)`. Table 1 shows the service specification that may be extracted from the two example methods.

Table 1: Service specification that is extracted from the external methods in Figure 1

Method Name	Service Name	HTTP Function	Input Parameters	Output Parameters
led_control	led_control	POST	status	
getTemperature	getTemperature	GET		temp

2.4 IoT Services

An IoT device may have multiple functionalities. For instance, an indoor sensor may sense both temperature and humidity. A functionality may be implemented by one or more external methods. In our approach, each functionality of an IoT device is transformed to an IoT service, which is hosted on the cloud platform. The cloud platform may use various networking protocols to exchange data with IoT devices, such as MQTT [12]. MQTT is a lightweight publish-subscribe messaging protocol designed for exchanging real-time IoT data.

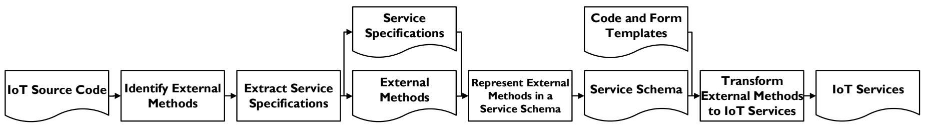
3 Overview of Our Approach.

In this section, we present our approach to automatically generate IoT services from IoT applications. Figure 2 shows an overview of our approach. Our approach has four activities. Each activity is explained in a subsection below.

[1] https://www.hackster.io/

[2] https://www.hackster.io/user3424878278/pool-fill-control-119ab7

[3] https://www.hackster.io/dexterindustries/add-a-15-display-to-the-raspberry-pi-b8b501



IoT Source Code
Identify External Methods
Extract Service Specifications
Service Specifications
External Methods
Represent External Methods in a Service Schema
Code and Form Templates
Service Schema
Transform External Methods to IoT Services
IoT Services

Fig. 2: An overview of our approach

3.1 Identifying External Methods

To save developers' effort on manually finding code methods that should be transformed, we analyze the source code of IoT applications written in Python to investigate whether external methods can be automatically identified. We choose the Python language, since it is suitable for developing IoT applications due to its portability and easy-to-learn syntax [21]. Although our approach is language-independent, we use Python examples to explain our approach implementation. We explain the steps that are involved in this code analysis below.

STEP1: Parsing Source Code of Methods. To identify external methods, we first analyze the Abstract Syntax Tree (AST) of the source code. An AST is a tree structure that represents the syntax of the source code. Each node in the tree describes a construct (*e.g.*, method name) that is present in the source code. We traverse the tree to identify the following constructs in a method:

– *method name, e.g.*, `getTemperature` shown in Figure 1b.
– *input variables, e.g.*, `status` shown in Figure 1a.
– *returned variables, e.g.*, `temp` shown in Figure 1b.
– *method calls in a method body, e.g.*, `turn_led_on()` shown in Figure 1a.
– *if-else statements, e.g.*, `if status == "ON"` shown in Figure 1a.

STEP2: Filtering Internal Methods. An internal method can be identified based on its extracted constructs. Methods with the following *internal features* (IF) are considered as internal methods. Internal methods are filtered out and are not investigated further.

– ***IF1****: method name containing the keywords "init, setup, debug, test".* Method names containing the keywords "init" and "setup" are initialization methods and are used to configure the initial settings, *e.g.*, setting the voltage level of GPIO pins. Method names containing the keywords "debug" and "test" are testing methods, which are used to test the different functionalities of an IoT device. Such testing methods are internal methods in an IoT device.
– ***IF2****: method name starting with "_".* The leading underscore in a method name denotes that the method is for internal use or reserved for the programming language (*e.g.*, an *__init__* method) [1].
– ***IF3****: methods that are called within internal methods or defined in internal files.* File names containing the keywords "init, setup, debug, test" or starting with "_" are internal files. Methods that are called within internal methods or defined in internal files are used for initialization and testing.

STEP3: Processing Method Names. A method name may convey the intent of the method, which can be used to distinguish external methods from internal methods. To identify the semantics of method names, we use the following steps to normalize these names. We split CamelCase words (*e.g.*, *getTemperature* is split into *get* and *temperature*). We remove the punctuation, *e.g.*, "_" and "-". We also remove the suffixes that contain numbers (*e.g.*, *led1* is normalized into *led*). Finally, we remove stop words (*e.g.*, "a", "the" and "is"). We use natural language processing (NLP) techniques to identify the part-of-speech (POS) tag of each word. For example, "get" is tagged as a verb and "temperature" is tagged as a noun. Finally, we perform word stemming to find the root words (*e.g.*, "reduced", "reducing" and "reduces" are normalized to "reduce"). These words are used to extract features for identifying external methods.

STEP4: Extracting Features for External Methods. We extract the following *external features* (EF) based on the constructs of the methods that are identified in STEP 1.

– ***EF1****: method calls.* If the methods that are called within a method body contain *send* related keywords in their names, *i.e.*, "push, post, publish, send, notify", these methods likely send data to the cloud, *e.g.*, `send(temperature, url)` and are considered as external methods.
– ***EF2****: if-else statements.* In case a method contains if-else statements that react to the input variables of the method when receiving commands from the cloud, such a method has a high probability of being an external method. For example, the `led_control` method in Figure 1a contains if-else statements that react to changes in the `status` variable.
– ***EF3****: semantic of verbs.* Verbs in method names may represent the action that is performed in a method. For instance, *control* and *get* are the verbs in the examples of Figure 1. We identify the semantic of verbs to infer external methods. For example, if a verb has keywords that are related to sending and receiving messages (*i.e.*, "push, post, publish, send, notify, subscribe, get, sense, set, receive, control"), we infer that its respective method transmits data to the cloud. These methods are likely external methods.
– ***EF4****: semantic of nouns.* Nouns in method names denote the objects of interest of these methods, *e.g.*, *led* and *temperature* are nouns in the examples of Figure 1. If these nouns match with IoT service names, their respective method is likely an external method. We identify IoT services by using the iotlist.co[4] website. This website lists various IoT devices, *e.g.*, security cameras and smart lights. For an IoT device, we manually extract their functionalities, each one corresponding to an IoT service name. For example, the Elgato Eve Room Wireless Indoor Sensor[5] will have the *sense air quality, sense temperature* and *sense humidity* IoT service names. In total, we extracted 190 IoT service names. Next, we use the approach described in STEP 3 to extract nouns from the extracted IoT service names. We form a bag of words containing the nouns and match them with the nouns that we find in method names (see STEP 3). For instance, the Wireless Indoor Sensor has a bag containing the *air, quality, temperature* and *humidity* words that we match with the *temperature* word in the `getTemperature` method (see Figure 1b).

[4] http://iotlist.co/
[5] https://www.elgato.com/en/eve/eve-room

In our approach, we assume that a method is an external method if it has at least two of the features that we identify in STEP 4. For example, the method in Figure 1a is an external method, since it has the *if-else statements* (*i.e.*, *EF2*) and *semantic of nouns* (*i.e.*, *EF4*) features.

3.2 Extracting Service Specifications

Based on the analyzed external methods, we extract the service specifications for their respective IoT services (see Table 1), *i.e.*, *service name, HTTP function* and *input (or output) parameters.*

We use the method name as the service name. For instance, *led_control* is the service name for the method in Figure 1a. Then, we use the *external features* described in STEP 4 to distinguish HTTP GET and POST functions. Each HTTP function is associated with two *external features.* Among these *external features*, we split the *semantic of verbs* into *semantic of send* and *semantic of receive* for GET and POST functions, respectively. We explain the details below.

– *HTTP GET:* is associated with the *semantic of send* and *method calls* features. The *semantic of send* denotes that a verb in a method name contains *send* related keywords, *i.e.*, "push, post, publish, send, notify, get, sense". Such methods send data to the cloud, so that GET-based IoT services can identify and retrieve this data.
– *HTTP POST:* is associated with the *semantic of receive* and *if-else statements* features. The *semantic of receive* denotes that a verb in a method name contains *receive* related keywords, *i.e.*, "set, receive, control, subscribe". An IoT device receives commands from POST-based IoT services.

To determine which HTTP function should be associated with an external method, we count the number of features that belong to an external method. If an external method has a given feature, that feature has a counter of 1 (one). We derive a score for the HTTP GET function (*i.e.*, S_{get}) using Equation 1 and a score for the HTTP POST function (*i.e.*, S_{post}) using Equation 2.

$$S_{get} = C_{semantic\ of\ send} + C_{method\ calls} \tag{1}$$

$$S_{post} = C_{semantic\ of\ receive} + C_{if-else\ statements} \tag{2}$$

where $C_{semantic\ of\ send}$, $C_{method\ calls}$, $C_{semantic\ of\ receive}$ and $C_{if-else\ statements}$ denote the counters for the respective features.

We use the S_{get} and S_{post} scores to determine whether the HTTP function should be GET or POST, i.e., whichever has the highest value. In case S_{get} is equal to S_{post}, we calculate the fan-in and fan-out of an external method [22]. Fan-in represents the number of input variables of an external method, while fan-out denotes the number of returned variables of an external method. When a fan-in to fan-out ratio is larger than one, the POST function is chosen, since such a ratio indicates that an external method is written to receive data (see `led_control` in Figure 1a). The GET function is chosen otherwise.

Finally, the parameters of IoT services are extracted based on the identified HTTP functions. For example, the returned variables of a GET-based external method are extracted as the output parameters of the corresponding IoT service. Comparatively, the input variables of a POST-based external method are extracted as the input parameters of the corresponding IoT service. As an example, the `status` variable of the POST-based `led_control` method (shown in Figure 1a) is extracted as a service input parameter.

3.3 Representing External Methods in a Service Schema.

To transform external methods to IoT services, we need a structured data format that describes the extracted service specifications of IoT services. We design a service schema using the Web Ontology Language (OWL) [2]. In the service schema, the identified service name, HTTP function and parameters of a service specification are prefilled. A developer may validate, modify and complete the service schema. Figure 3 shows how we use OWL to define our service schema.

The service schema is composed of four main components: *classes, individuals, relations* and *attributes.* A *class* represents a group of objects with similar properties. For example, an *IoT device* is a class. A *relation* is used to connect the components of our service schema (e.g., an *IoT service hasOperations*). A *class* can be inherited by *sub-classes.* For instance, a *reading* operation, which is used to get the latest value of a sensor, is a sub-class of *operation.* An *individual* is an instance of a class. Finally, *attributes* declare the properties of a class. For instance, the *IoT device* class has the *device type* and *device id* attributes. The *device type* groups a number of IoT devices that provide similar functionalities. For example, *temperature sensor* may be a device type. The *device id* attribute is unique for each IoT device and is used to distinguish one IoT device from another. The MAC address of an IoT device can be used as a device id.

A functionality of an IoT device publishes a single stream of scalar values (*e.g.*, temperature values) to a channel on the cloud [4]. The stream of scalar values is considered as a resource of an IoT service. This resource is stored in a resource database on the cloud. An IoT service identifies its resources using the *service name, device type* and *device id* attributes. An IoT service provides multiple operations to perform different actions onto a resource. For instance, the IoT service "sense temperature" can obtain the latest reading of the temperature and modify the frequency at which the temperature should be sensed. We identify six operations of IoT services based on the approach proposed by Haggerty *et al.* [4], *i.e.*, *reading*, *profile*, *sampling parameter*, *formatting*, *status* and *context.* Each external method falls in one of the operations specified in Table 2. The *reading* operation is used to get the latest value of a resource. This operation listens to an IoT service's resource until a new value of that resource is received.

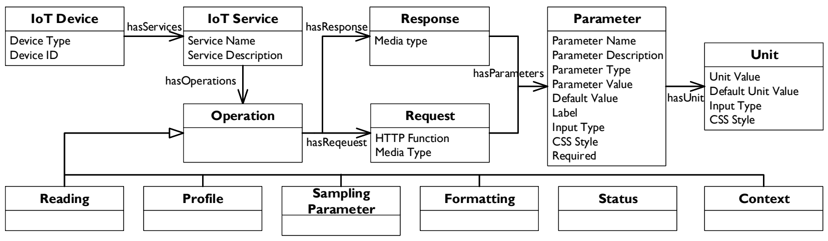


Fig. 3: The service schema

```
<Device Type>/<Device ID>/<Service Name>/<Operation Name>
```

Fig. 4: The URL schema for accessing an operation.

Then, the listened value and a timestamp of the value update are returned to end-users. The *status* operation returns the state of a given IoT service (*e.g.*, whether it's on or off). For actuators, an end-user may send a POST request to the *status* operation, which changes the physical state of an IoT device (*e.g.*, to turn on the light). An operation is identified by the URL pattern (see Figure 4).

IoT devices with the same *device type* value correspond to one unique service schema that is used to describe their respective IoT services. We use the service schema to instantiate IoT services as we describe in Section 3.4.

Table 2: A summary of available operations for an IoT service

Operation Name	**HTTP Function**	**Device Class**	**Description**	**Example**
Reading	GET	sensor	the latest reading	temperature
Profile	GET	sensor	a number of recent history readings	history temperature readings
Sampling Parameter	GET/POST	sensor	the sampling frequency of sensing	100Hz
Formatting	GET/POST	sensor	the unit of the sensed value	°C, °F
Status	GET/POST	sensor/actuator	the state of the IoT service	turn light on/off
Context	GET/POST	sensor/actuator	the location of the measurement	the location that the temperature is being sensed

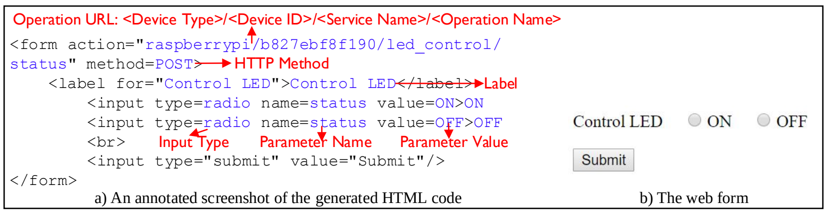
3.4 Transforming External Methods to IoT Services

In this section, we describe how our approach automatically transforms external methods to IoT services.

STEP1: Generating Web Forms. Since end-users may not be familiar with SOA, it is important to provide friendly user interfaces for accessing and controlling IoT services. In this regard, our approach automatically generates web forms by using our proposed service schema and form templates. A template uses the data of a service schema to generate text output, *e.g.*, source code or HTML forms. These generated forms are used to send POST requests to IoT services.

We design our form templates using the FreeMarker template engine.[6] The essential components of a web form are the HTTP function, the operation URL and the parameters to be filled by end-users. We traverse the parameters in our service schema to identify which ones have an *input type* attribute. The *input type* attribute can assume one of the HTML input elements, *i.e.*, *text, radio* and *select.* The *parameter value* attribute (see Figure 3) defines the available options of a parameter, which end-users can choose, *e.g.*, ON or OFF. A developer may define a CSS style for an input parameter using the *CSS style* attribute. Figure 5 shows an HTML form example for controlling a led. Once an end-user clicks on the "Submit" button, a POST request is submitted to the operation URL.

[6] http://freemarker.org/



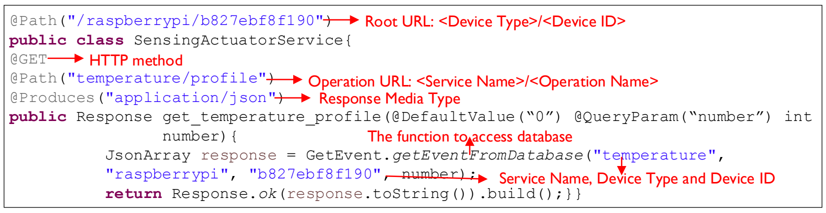
```
<form action="raspberrypi/b827ebf8f190/led_control/
status" method=POST>
    <label for="Control LED">Control LED</label>
        <input type=radio name=status value=ON>ON
        <input type=radio name=status value=OFF>OFF
        <br>
        <input type="submit" value="Submit"/>
</form>
```

Operation URL: <Device Type>/<Device ID>/<Service Name>/<Operation Name>
HTTP Method
Label
Input Type
Parameter Name
Parameter Value
Control LED
ON
OFF
Submit
a) An annotated screenshot of the generated HTML code
b) The web form

Fig. 5: An annotated screenshot of a web form that is used to control a led. The blue text highlights the data that is extracted from our service schema.

STEP2: Instantiating IoT Services. Our approach automatically generates source code to instantiate IoT services using the proposed service schema and code templates. The instantiated IoT services follow the Jersey[7] syntax standard.

To instantiate an IoT service, a code template needs to be filled with the information from a service schema. The required information are the HTTP function, an operation URL, a request media type, a response media type and the filled parameters from a web form. We provide code templates for each kind of operation. In a *reading* template, a function is provided to listen to a resource of an IoT service, which then responds end-users with the real-time value. The resources of a given IoT service are located by the *service name*, *device type* and *device id* that are extracted from the URL of the respective service request (see Figure 4). Once the *profile* operation is requested, the IoT service fetches the last N values of a resource from the database. The N value is specified by end-users as a URL parameter. We also build databases on the cloud for each of the other four operations, *i.e.*, *sampling parameter, formatting, status* and *context*. A GET request of an operation locates the respective database and retrieves the data value. Figure 6 shows an example of an instantiated *profile* operation.



```
@Path("/raspberrypi/b827ebf8f190")
public class SensingActuatorService{
@GET
@Path("temperature/profile")
@Produces("application/json")
public Response get_temperature_profile(@DefaultValue("0") @QueryParam("number") int
                number){
        JsonArray response = GetEvent.getEventFromDatabase("temperature",
        "raspberrypi", "b827ebf8f190", number);
        return Response.ok(response.toString()).build();}}
```

Root URL: <Device Type>/<Device ID>
HTTP method
Operation URL: <Service Name>/<Operation Name>
Response Media Type
The function to access database
Service Name, Device Type and Device ID

Fig. 6: An annotated screenshot of the *profile* operation, which obtains the temperature. The blue text highlights the data that is extracted from our service schema.

For POST-based operations (*e.g.*, POST status of light), the filled parameters (*e.g.*, ON) must be sent to the respective IoT device. We traverse the parameters of a service schema to identify which parameters end-users should fill. Instantiated POST-based operations use their parameter names (*e.g.*, status) as variables that will retrieve the values that are filled in web forms.

Once end-users invoke an instantiated operation, the generated source code of that operation is accessed by the operation URL and the HTTP function. The data that is transmitted between an IoT device and the cloud follows the JSON standard (*i.e.*, JavaScript Object Notation), a lightweight data-interchange format [3].

[7] https://jersey.java.net/

4 Case Study.

We conduct a case study to evaluate our approach. In this section, we introduce the setup of our case study and we present the obtained results.

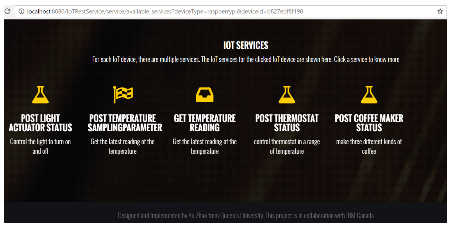
4.1 Case Study Setup

To test the effectiveness of our approach on identifying external methods, we analyze IoT applications written in Python. We collect the source code of IoT projects in the "Raspberry Pi" category on the hackster.io website.[8] The Raspberry Pi is a credit-card-sized embedded device, which is widely used to develop IoT solutions for home and industrial automation.

Table 3: The distribution of projects and python methods in each domain. *Avg LOC* denotes the average lines of code for each project.

Domain	**# Projects**	**# Methods**	**Avg LOC**	**Domain**	**# Projects**	**# Methods**	**Avg LOC**
Living	41	7,035	6,349	Environmental Sensing	24	763	553
Transportation	10	387	756	Health	21	599	728
Entertainment	41	4,660	1,781	Security	22	1,122	870
Communication	18	2,612	3,891	**Total**	177	17,178	2,520

In total, we collect 1,039 projects, of which 177 contain python methods. We collect a total of 17,178 python methods from these projects. The collected IoT projects have different domains, *e.g.*, living (*e.g.*, light control), communication (*e.g.*, radio receiver) and transportation (*e.g.*, parking system). Table 3 summarizes our collected data. We built a prototype tool as a proof of concept for our approach.


localhost:8080/IoTRestService/service/available_services?deviceType=raspberrypi&deviceId=b827ebf8f190

IOT SERVICES

For each IoT device, there are multiple services. The IoT services for the clicked IoT device are shown here. Click a service to know more

POST LIGHT ACTUATOR STATUS
Control the light to turn on and off

POST TEMPERATURE SAMPLINGPARAMETER
Get the latest reading of the temperature

GET TEMPERATURE READING
Get the latest reading of the temperature

POST THERMOSTAT STATUS
control thermostat in a range of temperature

POST COFFEE MAKER STATUS
make three different kinds of coffee

Designed and Implemented by Yu Zhao from Queen's University. This project is in collaboration with IBM Canada

Fig. 7: A screenshot of our tool that shows the available operations of IoT services to end-users

Our tool automatically analyzes IoT applications and generates the corresponding IoT services based on the identified external methods. We use the Raspberry Pi 3 Model B as our IoT device (denoted as *RPi*). The IoT device has a quad-core processor running at 1.2GHz. We use the IBM cloud platform, which uses the MQTT protocol to communicate with IoT devices. Since we are not allowed to build customized IoT services in such a commercial cloud platform, we use our approach to generate IoT services in our local server (see Section 3.4). Our server transmits data of IoT devices with the IBM cloud platform using the MQTT protocol. Figure 7 shows a screenshot of our prototype tool. An end-user may click on an operation to send a GET request or retrieve a web form to submit a POST request. Our case study answers the following research questions:

RQ1. How effective is our approach to identify external methods and extract service specifications?

[8] https://www.hackster.io/

RQ2. How accurate is our approach to generate IoT services?

The first author manually evaluates all the results in our case study. Our evaluator has three years' experience on building RESTful services for the service-oriented architecture.

4.2 RQ1: How effective is our approach to identify external methods and extract service specifications?

To measure the effectiveness of our approach, we randomly sample 376 methods from the extracted 17,178 python methods with a 95% confidence level and a 5% confidence interval [25]. We apply our approach (as described in Section 3.1) to identify external methods and extract service specifications from the sampled 376 methods. We use precision and recall as shown in Equations 3 and 4 to evaluate our approach. *Precision* measures the ratio of correctly retrieved external methods (or service specification parts) from the set of external methods (or service specification parts) that are retrieved by our approach [24]. On the other hand, *recall* measures the ratio of external methods (or service specification parts) from the dataset that our approach could retrieve [24].

$$Precision = \frac{\{relevant\ items\} \cap \{retrieved\ items\}}{\{retrieved\ items\}} \tag{3}$$

$$Recall = \frac{\{relevant\ items\} \cap \{retrieved\ items\}}{\{relevant\ items\}} \tag{4}$$

Our results reveal that our approach has an average precision of 75% and a recall of 72% for identifying external methods. As for service specifications, our approach has an average precision of 82% and a recall of 81%. The main reasons for the misidentified external methods and service specifications are the following: (1) We are not able to extract semantic meanings from method names. For instance, the method `getdoorstatus` is an external method to get the door status. However, we could not find the *semantic of nouns* and *semantic of verbs* because this name does not follow the CamelCase pattern. (2) Internal methods that transmit messages within an IoT device may have *external features.* For example, the method `SendParameter` is an internal method that sends parameters using the I2C (Inter-Integrated Circuit) protocol. However, we identify such a method as an external method, since it contains the *semantic of verbs* and *if-else statements* features. (3) The input (or output) parameters are defined in the code method body. For instance, the method `get_ph_reading` has a service parameter *ph_value.* However, the method uses a print function to display the parameter, rather than returning it.

4.3 RQ2. How accurate is our approach to generate IoT services?

Our approach uses a service schema to generate IoT services on the cloud platform. The accuracy of transforming external methods to IoT services is what represents the practical usefulness of our approach. To evaluate the accuracy of generating IoT services, we use the 190 extracted IoT services described in Section 3.1. We design external methods on *RPi* depending on the type of an IoT service. For example, we design four possible external methods for an IoT service that is generated for a sensor. These methods are: *reading, sampling parameter, formatting* and *context.* As for IoT services generated for actuators, we design two external methods: *status* and *context.* We do not design an external method for the *profile* operation, since a *profile* operation is instantiated to fetch a resource from a resource database. We use the approach described in Section 3.3 to generate service schemas for the designed external methods. We automatically generate IoT services using our approach (see Section 3.4). Equation 5 shows how we measure the accuracy of our approach.

$$Accuracy = \frac{\{\#correctly\ generated\ IoT\ services\}}{\{\#IoT\ services\}} \tag{5}$$

The accuracy is the ratio of the number of correctly generated IoT services to the total number of IoT services. Since an IoT service is composed of several operations, we evaluate whether an operation is correctly instantiated. A GET-based operation is correctly instantiated if, for example, a GET request for the *temperature reading* operation returns the values that match the temperature values sent from *RPi.* A POST-based operation is correctly instantiated if, for example, the external method on *RPi* that is used for receiving light status receives the commands from the *light status* operation. An IoT service is correctly generated when all the operations of such an IoT service are correctly instantiated. Our approach achieves an accuracy of 96% (182 out of 190 IoT services) when generating IoT services. Nonetheless, our approach fails to generate IoT services regarding streaming media. A streaming media IoT service constantly delivers and presents multimedia, *e.g.*, video and audio, to end-users. We do not find support in the IBM cloud platform for streaming media of IoT devices.

5 Related Work

We summarize the related work on the service-oriented architecture for IoT devices and code analysis.

5.1 Service-oriented architecture for IoT devices

Service-oriented architecture (SOA) [22] is widely used to represent functionalities of IoT devices [13]. Haggerty *et al.* [4] and Guinard *et al.* [10] design IoT services using the RESTful paradigm. Priyantha *et al.* [15] propose an approach to reduce the resource consumption when running IoT services on IoT devices. The aforementioned approaches build IoT services directly on the resource constrained IoT devices. In contrast to these approaches, we use the cloud platform to run the IoT services. SOCRADES [9][20] describe IoT services using the Device Profile for Web Services (DPWS), a service description language. Other approaches [7][8][27] model IoT services using ontology languages, such as

OWL-S. These service models are used to aid the service discovery and selection. Different from the existing approaches, our approach designs the service schema for the automatic service generation to relieve extra effort to build SOA.

5.2 Code analysis

Code analysis is widely used to aid software understanding and maintenance. For example, Eaddy *et al.* [5] and Robillard *et al.* [18] analyze the dependency and relationship of program elements (*e.g.*, class and method) to identify the source code that is related to a maintenance task. Eisenbarth *et al.* [6] conduct static and dynamic code analysis to focus on the source code that is related to system features. Zhou *et al.* [26] and Wong *et al.* [23] locate source code files that are related to faults in bug reports. Pollock *et al.* [14] use natural language processing techniques to understand the semantic meanings of literals, identifiers and comments to aid the source code searching. Shabtai *et al.* [19] extract features from the source code to classify Android applications. Unlike these approaches, our approach conducts a static code analysis on the method level to identify external methods and extract service specifications for IoT services.

6 Conclusion

To enable the integration of multiple IoT devices in a uniform environment, we provide an approach that automatically transforms functionalities from IoT devices to SOA based IoT services. We also automatically generate web forms for end-users to have a friendly experience when interacting with IoT services. We use the designed service schema and templates to generate IoT services. Our case studies show that we can identify external methods from IoT applications with a precision of 75% and a recall of 72%. We can also extract service specifications from these external methods with a precision of 82% and a recall of 81%. Our approach can generate IoT services with an accuracy of 96%.

In future work, we plan to extend the implementation of our approach to other popular programming languages, such as Java and JavaScript. We also plan to ask developers to use and verify our prototype tool.

References

1. Style guide for python code, `https://www.python.org/dev/peps/pep-0008/`
2. Bechhofer, S.: Owl: Web ontology language. In: Encyclopedia of Database Systems, pp. 2008–2009. Springer (2009)
3. Crockford, D.: The application/json media type for javascript object notation (json) (2006)
4. Dawson-Haggerty, S., Jiang, X., Tolle, G., Ortiz, J., Culler, D.: smap: a simple measurement and actuation profile for physical information. In: sensys (2010)
5. Eaddy, M., Aho, A.V., Antoniol, G., Guéhéneuc, Y.G.: Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In: ICPC. pp. 53–62. IEEE (2008)

6. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. TSE 29(3), 210–224 (2003)
7. Eisenhauer, M., Rosengren, P., Antolin, P.: Hydra: A development platform for integrating wireless devices and sensors into ambient intelligence systems (2010)
8. Escobedo, E.P., Prazeres, C.V., Kofuji, S.T., Teixeira, C.A., da Graça Pimentel, M.: Secoas: An approach to develop semantic and context-aware available services. WebMedia 7, 1–8 (2007)
9. Guinard, D., Trifa, V., Karnouskos, S., Spiess, P., Savio, D.: Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. TSC 3(3), 223–235 (2010)
10. Guinard, D., Trifa, V., Pham, T., Liechti, O.: Towards physical mashups in the web of things. In: INSS. pp. 1–4. IEEE (2009)
11. Hachem, S., Teixeira, T., Issarny, V.: Ontologies for the internet of things. In: Proceedings of the 8th Middleware Doctoral Symposium. p. 3. ACM (2011)
12. Hunkeler, U., Truong, H.L., Stanford-Clark, A.: Mqtt-s—a publish/subscribe protocol for wireless sensor networks. In: comsware. pp. 791–798. IEEE (2008)
13. Issarny, V., Bouloukakis, G., Georgantas, N., Billet, B.: Revisiting service-oriented architecture for the iot: A middleware perspective. In: ICSOC. pp. 3–17 (2016)
14. Pollock, L., Vijay-Shanker, K., Hill, E., Sridhara, G., Shepherd, D.: Natural language-based software analyses and tools for software maintenance (2013)
15. Priyantha, N.B., Kansal, A., Goraczko, M., Zhao, F.: Tiny web services: design and implementation of interoperable and evolvable sensor networks (2008)
16. Rao, B.P., Saluia, P., Sharma, N., Mittal, A., Sharma, S.V.: Cloud computing for internet of things & sensing based applications. In: ICST. pp. 374–380 (2012)
17. Richardson, L., Ruby, S.: RESTful web services. O'Reilly Media, Inc. (2008)
18. Robillard, M., Murphy, G.C.: Concern graphs: finding and describing concerns using structural program dependencies. In: ICSE. pp. 406–416. IEEE (2002)
19. Shabtai, A., Fledel, Y., Elovici, Y.: Automated static code analysis for classifying android applications using machine learning. In: CIS. pp. 329–333 (2010)
20. de Souza, L., Spiess, P., Guinard, D., Köhler, M., Karnouskos, S., Savio, D.: Socrades: A web service based shop floor integration infrastructure (2008)
21. Tanganelli, G., Vallati, C., Mingozzi, E.: Coapthon: easy development of coap-based iot applications with python. In: WF-IoT. pp. 63–68. IEEE (2015)
22. Upadhyaya, B., Zou, Y., Xiao, H., Ng, J., Lau, A.: Migration of soap-based services to restful services. In: WSE. pp. 105–114. IEEE (2011)
23. Wong, C.P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., Mei, H.: Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: ICSME. pp. 181–190. IEEE (2014)
24. Zhao, Y., Wang, S., Zou, Y., Ng, J., Ng, T.: Mining user intents to compose services for end-users. In: ICWS. pp. 348–355. IEEE (2016)
25. Zhao, Y., Zhang, F., Shihab, E., Zou, Y., Hassan, A.E.: How are discussions associated with bug reworking?: An empirical study on open source projects. In: ESEM, ACM. p. 21 (2016)
26. Zhou, J., Zhang, H., Lo, D.: Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In: ICSE (2012)
27. Zhu, W., Zhou, G., Yen, I.L., Bastani, F.: A pt-soa model for cps/iot services. In: ICWS. pp. 647–654. IEEE (2015)