JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 38, 317-341 (2022)
DOI: 10.6688/JISE.202203_38(2).0003

A Novel Approach to Automate IoT Testing of Gateways and Devices*

CHIEN-HUNG LIU[1], WEN-YEW LIANG[2], MING-YI TSAI[3],
WEI-CHE CHANG[4] AND WOEI-KAE CHEN[1]
[1]*Department of Computer Science and Information Engineering*
National Taipei University of Technology
Taipei, 106 Taiwan
[2]*ADLINK Technology Inc., New Taipei City, 235 Taiwan*
[3]*NextDrive Co., Taipei, 115 Taiwan*
[4]*Memopresso Inc., Taipei, 110 Taiwan*
E-mail: cliu@ntut.edu.tw; william.wyliang@gmail.com; coopldh@gmail.com;
wayne265265@gmail.com; wkchen@ntut.edu.tw

The software (firmware) testing for Internet of Things (IoT) usually involves multiple communication protocols and different hardware devices. Together with complex user scenarios/environments and frequently updated firmware, it can be time-consuming and error-prone to build test environments and conduct the testing. To alleviate these problems, this paper presents an approach and an automated IoT Testing Tool, called IoT3, which automatically builds test environments, executes tests for IoT gateways, and performs system compatibility testing. For gateway testing, IoT3 supports keyword-driven testing method, facilitating the development and maintenance of test scripts. In addition, a mock device method is proposed to simulate the changes of environment conditions, such as temperature, humidity, and the packet loss of Bluetooth connections. For compatibility testing, IoT3 allows a test engineer to select a target environment, a particular version combination of the IoT devices (gateways, sensors, or Apps), and performs a full compatibility testing for the devices. To evaluate the effectiveness of the approach, an industry case study was conducted. The results indicate that, as compared to manual testing, using the proposed approach (IoT3) can save testing time, reduce human efforts, improve test coverage, and also detect more defects.

Keywords: IoT testing, software testing, compatibility testing, gateway, mock device, firmware

1. INTRODUCTION

The Internet of Things (IoT) devices have experienced a blooming development in recent years. It is estimated that there will be 41.6 billion connected IoT devices in 2025 and 79.4 zettabytes (ZB) of data will be generated by these devices [1]. As IoT devices and applications grow rapidly, testing the quality and reliability of IoT devices becomes increasingly important. In particular, before releasing a new version of a device, the test engineer (hereafter called tester) not only needs to conduct a thorough testing to validate the correctness of the new version, but also needs to ensure that the new version is compatible to all versions of the other cooperating devices. This is costly and very time consuming.

Received October 2, 2020; revised December 1, 2020; accepted February 1, 2021.
Communicated by Chu-Ti Lin.
* This research was partially supported by the Ministry of Science and Technology, Taiwan, under contract No. MOST108-2221-E-027-036 and 109-2221-E-027-079.

This paper studies the software/firmware testing of IoT gateways and devices for applications such as smart home where IoT devices are inter-connected to IoT gateways, allowing automatic measurement and control of household environments and/or home appliances. A typical setup is depicted in Fig. 1 where devices such as temperature, humidity, and motion sensors are connected to an IoT gateway via Bluetooth or USB. The gateway processes the data/events received from the sensor, triggers additional events/alerts using a set of rules defined by the user, and also uploads the information to the cloud. The user accesses/manipulates the gateway through a (mobile) App, which allows the user to view the sensor information and setup the connections between the gateway and the devices. Note that, in this paper, the IoT gateway, sensor devices, mobile phones (or Apps), and Wi-Fi AP shown in Fig. 1 are all considered as IoT devices.

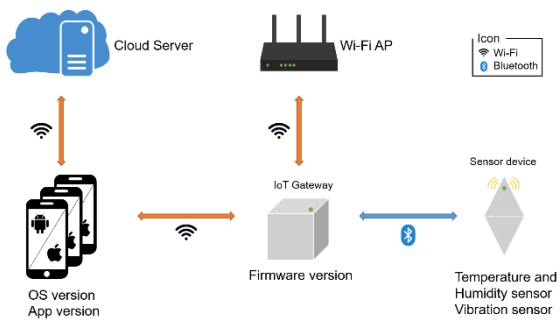

Fig. 1. A typical setup of IoT applications.

The challenge of testing the IoT gateways and devices is that this is a type of system integration testing. The issues that should be considered include:

(a) **How to control testing environment?** While receiving sensor information from an IoT device, the gateway must analyze the information and perform corresponding actions. For example, if the sensor reports an abnormally high temperature, the gateway needs to alarm the user. However, it is not economical to actually raise the room temperature solely for the testing of temperature change.

(b) **How to control wireless signal strength?** Testing the connectivity between the gateway and the device under different Bluetooth signal strength is vastly important. However, it is unrealistic to control the signal strength simply by moving the gateway and device apart.

(c) **How to reduce test cost?** To meet market demands rapidly, the software/firmware of both the gateway and devices is updated frequently. Such updates dictate repeated testing, and consequently the test cost, the human efforts involved in the testing, becomes expensive if the testing is not highly automated.

(d) **How to shorten time to market?** When a new firmware is to be released, testing the compatibility between the App, gateway, device, and cloud service become a huge job. In particular, testing the compatibility between all different software/firmware/hardware versions is not only expensive but also time consuming.

To overcome the above issues and improve the quality and testability of testing IoT gateway and devices, this paper proposes a novel approach and an IoT testing tool called

IoT3 which builds test environments, executes tests, and report test results automatically. The testing is centered on both the gateway and the compatibility between IoT devices. For gateway testing, IoT3 supports keyword-driven testing method, facilitating the development and maintenance of test scripts. In addition, to increase the controllability and observability as well as to reduce test cost, a mock device method is proposed. The mock device can simulate the changes of environment conditions, such as temperature, humidity, and the packet loss of Bluetooth connections. For compatibility testing, IoT3 allows a tester to select one or more target environments for testing. A target environment is a particular version combination of the IoT devices (gateways, sensors, or Apps). IoT3 automatically prepares the target environments and performs compatibility testing of the devices under such environments. We conduct an industry case study to evaluate the effectiveness of the proposed approach. The results indicate that, as compared to manual testing, IoT3 can save testing time, reduce human efforts, improve test coverage, and also detect more defects.

The rest of this paper is organized as follows. Section 2 briefly reviews existing studies related to our work. Section 3 presents the proposed approach. Section 4 describes the design and use of IoT3. Section 5 reports the evaluation results. The concluding remarks and future work are given in Section 6.

2. RELATED WORK

The exponential growth of IoT applications has attracted much attentions on the quality and reliability of IoT-enable products. The followings briefly review existing studies on IoT testing. Based on the problem addressed, these studies are further categorized into two areas: IoT testing challenges/approaches/experiences and IoT testing platforms/frameworks/tools.

• IoT testing challenges/approaches/experiences

Taivalsaari and Mikkonen [2] described the new challenges posed by IoT for software developers. Particularly, the large number of devices, dynamic topologies, unreliable connectivity, device heterogeneity and (sometimes) invisibility can introduce challenges for debugging and testing IoT systems. Although testing an individual device might be reasonably easy, it can be very challenging to test an entire IoT system containing hundreds or thousands of devices deployed in a complex environment.

Kanstrén *et al.* [3] presented their experiences in building IoT test environments at different levels of the IoT stack and discussed related test architectures. For example, on the sensor level, modelling and simulating various forms of sensor data from real IoT sensor devices can be essential for generating realistic test input data. On the gateway level, it is critical to verify if the gateway can handle different numbers of sensors, combinations of different IoT protocols, and data filtering and aggregation. On the network level, IoT specific protocols, network configurations, traffic patterns, and power usage can be the important properties to be tested.

Dias *et al.* [4] provided an overview on existing approaches, tools and methodologies for testing IoT systems. Specifically, they described and compared sixteen IoT testing tools that can be searched from Scopus and Google. These tools are compared in terms of varied factors, such as the IoT Layer that they focus on, the testing level that the tools can be

applied to, the test method and environment that the tools use, *etc.* They also found that these IoT testing tools still have limitations and further work must be pursued on the development of IoT testing solutions.

Bures *et al.* [5] conducted an industry survey on the IoT quality assurance and testing techniques among ten large international IoT solutions providers in various areas. The results indicate that the top three quality issues considered significant by the companies are the behavior of the IoT system under limited network connection, mutual compatibility (*i.e.*, interoperability) of the IoT devices, and testing problems caused by a large number of various system configurations. Based on the findings, they suggest that IoT-specific testing methods and automated integration testing of IoT solutions need to be explored further.

Hwang *et al.* [6] proposed an approach called AUTOCON-IoT to automate and scale the conformance testing for IoT applications. Particularly, the approach introduces a test triggering mechanism that can send standardized messages to request the IoT system under test to perform specified tasks and check its functional compliance automatically. The experimental results indicate that the conformance testing time can be greatly reduced as compared with hybrid and manual testing.

Brady *et al.* [7] proposed a method to emulate realistic IoT test environment for anomaly detection with the Network Emulator for Mobile Universes (NEMU). The NEMU is a tool that can manage a dynamic network of virtual machines. It can emulate IoT devices (*e.g.*, Raspberry Pi) by using QEMU, a widely used full-system open source machine emulator. The experimental results suggest that the emulated environment can have similar results as compared with the real one. However, the effort of building the IoT test environment can be reduced significantly.

• IoT testing platforms/frameworks/tools

Rosenkranz *et al.* [8] analyzed the testing challenges posed by IoT software and proposed a test system architecture for open-source IoT software. The architecture is composed of a central continuous integration (CI) broker and a test cluster. The test cluster consists of clients running on distributed computers. Each client can provide access to attached build environments and/or test platforms. Moreover, a platform abstraction layer is used to allow for test cases to interact with test platforms using one unified API so that test cases can be executed across different test platforms. The proposed architecture can be used for IoT interoperability testing.

Kim *et al.* [9] presented an automated service-oriented prospective framework, called IoT-TaaS, for testing IoT devices. The IoT-TaaS aims to resolve the coordination, costs, and scalability issues of traditional standards-based testing for IoT devices and support distributed interoperability testing, scalable conformance testing, and semantics validation testing. The architecture design as well as the plug-and-test implementation concept of IoT-TaaS are described. Moreover, the technological and systematic advancement of IoT-TaaS are also presented.

Palattella *et al.* [10] reported the F-Interop platform and its integrated tools for IoT interoperability testing. The F-Interop is a H2020 European research project [11, 12]. Specifically, it is a platform as a service (PaaS) that provides a set of online testing tools for IoT protocols. The users can connect to the F-Interop platform remotely and use the online tools to test compliance and interoperability of their IoT implementations against a variety

of protocol standards. With the F-Interop, the validation and standardization of IoT implementations can be facilitated and, hence, help reducing their time to market.

Pontes *et al.* [13] presented a pattern-based test automation framework called Izinto for integration testing of IoT systems. In particular, the framework supports a set of test patterns that are designed specifically for the IoT domain, such as the periodic reading test of sensors. These test patterns can be instantiated easily with minimal technical knowledge in order to create concrete test scenarios automatically for examining an IoT application. The proposed framework has been applied to the domain of ambient assisted living (AAL) to illustrate its usefulness.

Demirel *et al.* [14] described an interoperability testing platform called InterOpT for IoT systems based on the oneM2M standard, a global technical standard for Machine-to-Machine and IoT technologies. Specifically, the InterOpT consists of 5 main components including a test engine, statistics and analysis, reporting, test packages, and protocol modules. It can be used to validate if an IoT component is oneM2M compliance and facilitate the analysis of test results. Further, a test library has also been created in accordance with oneM2M standards to help the development of tests.

Another attempt was the earlier work of this paper [15], which briefly reported an IoT compatibility testing tool, called ICAT. This paper extends ICAT into IoT3 to cover a broader range of IoT gateway and device testing. The extension includes the discussion of the challenges, review of related work, description of the methodologies, and evaluation of the new case study. For completeness, the ICAT compatibility testing method is also included in paper.

As compared to the related work, the paper mainly focuses on the approach and supporting tool to automate IoT gateway testing and IoT system compatibility testing. The challenges of manually testing IoT gateways and devices are addressed. The approach to automatically build test environments, executes tests for IoT gateways, and performs system compatibility testing is presented. The design of a supporting tool, called IoT3, is described. The experiences of applying IoT3 to an industry case is also reported.

3. THE PROPOSED APPROACH

This section first elaborates the challenges of testing IoT gateways and devices briefly described in Section 1. Then, the proposed approach to alleviate the challenges by automating the testing of IoT gateways and devices is presented. After that, the approach for IoT compatibility testing is detailed.

3.1 The Challenges of Testing IoT Gateways and Devices

Fig. 2 shows an IoT gateway that is connected to various types of devices through Bluetooth or USB and can be controlled by a (mobile) App via Wi-Fi. As the gateway is the center of the application, it is critical to test whether the gateway is functionally correct and can work together with different types of devices under specified communication protocols, device configurations, and possible deployment environments. A typical scenario for testing the gateway is to setup the test environment and then test the gateway through the App manually. The setup includes resetting the gateway and devices, uploading the

firmware of the gateway, and pairing each device to the gateway, which can be quite time-consuming.

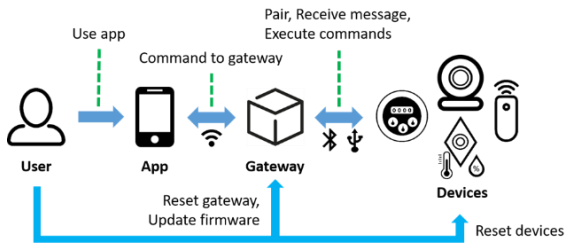

Fig. 2. A simple IoT system with a gateway and different devices.

The above manual testing scenario for IoT gateways can pose the following challenges.

• How to control testing environment?

The tester needs to create different environmental conditions so that the devices (sensors) can trigger the gateway for reactions, *e.g.*, triggering the reactions of abnormally high temperature, low battery, or high humidity. However, replicating a real-world environment to implement such tests can be very costly if not impossible. For example, suppose that the gateway shall send an alarm to the App if the temperature exceeds 50° in Celsius. To verify if the gateway does so, the tester may need to raise the room temperature so that the sensor device can capture the fluctuation of the room temperature and stimulate the gateway. Although such a testing can be done, it is time-consuming and costly. Thus, a better way of controlling the environmental conditions is necessary.

• How to control wireless signal strength?

The gateways and devices are usually connected together through a wireless network, such as Bluetooth. Therefore, the quality of wireless connection becomes a factor that needs to be considered carefully. To test whether the gateways and devices can work together correctly under different wireless signal strengths, the tester often needs to manually change the distances between the gateways and devices. However, this is often imprecise and difficult to control the connection quality. Consequently, the test results for different signal strengths can be affected substantially and may not be easy to reproduce.

• How to reduce test cost?

To satisfy the needs of customers and meet the market demands quickly, the software/firmware of IoT gateways and devices is often updated frequently in order to provide better, reliable and secure services. For each update of software/firmware, regression testing needs to be conducted thoroughly in order to ensure the quality of gateways and devices. However, according to the aforementioned test scenario, human effort is required to manually prepare and setup the test environment, conduct the test using the App, and record the test results. The effort can be very costly and thus test automation for IoT gateways and devices is very crucial.

• How to shorten time to market?

The above manual testing scenario can cause even more challenges in time to market if the compatibility of IoT devices is considered. The system compatibility testing needs to be conducted frequently and constantly whenever a new version of device is released or the firmware and/or software patches running on the device is updated due to any bug fixes, functional enhancements, and/or security reasons. However, conducting a thorough testing manually to ensure that the new or updated version is interoperable and compatible to all versions of the other devices is costly, error-prone, and very time consuming.

For instance, an IoT Vendor, called Vender A, releases new versions of IoT devices frequently in responding to the demanding and growing global IoT market. Fig. 3 depicts the active firmware versions of an IoT gateway manufactured by Vendor A after releasing a new firmware (v.29) of the gateway. On the 30th day of the release, 43% of the gateways were upgraded to v.29. However, v.14-v2.28 remained actively used. In other words, Vendor A cannot ignore the very existence of the earlier versions. Similar observations can also be found with Apps and sensors where several versions coexist at the same time. This creates a critical testing challenge for Vendor A - to release a new version of a device, the tester needs to conduct a thorough testing to ensure that the new version is compatible to all versions of the other devices.

For example, when v.29 is to be released, the new firmware should be tested against all versions of the Apps, sensors, *etc*. Worse still, at system level, every combination of the different versions of all IoT devices should be tested, because the change impact of one device could ripple through several devices. As an example, when Vendor A released v1.5.0 of their Android App in Google Play, due to an oversight in testing sensor connections (as shown in Fig. 1, the App does not connect to the sensors directly; therefore, such a testing was overlooked), the App crashed when some special sensor data were to be displayed. Vendor A was forced to create an emergency hot fix and released v1.5.1 in the next day.

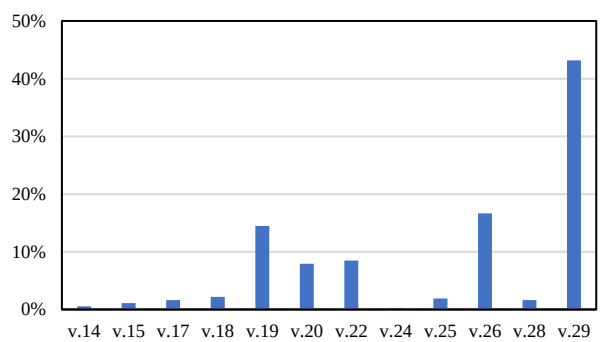

Fig. 3. The active firmware versions of a gateway device on the 30th day of the v.29 release.

Note that a critical issue for IoT system compatibility testing is that the number of target environments to be tested can be huge since the number of possible combinations for different versions of devices can be enormous. In the above example, suppose that there are 6 versions of sensors, 5 versions of gateway, 6+3 versions of Apps (6 and 3 versions of iOS and Android Apps, respectively), and 2 versions of Wi-Fi AP, a total of "6×5×

(6+3)×2=540" different target environments should be tested. This is not impossible, but is very time consuming and can be highly costly. Thus, it is crucial to speed up IoT system compatibility testing and help testers to find compatibility issues automatically.

3.2 The Approach for Testing IoT Gateways and Devices

To alleviate the challenges of testing IoT gateways and devices manually, a novel approach and a supporting tool called IoT3 are proposed. Particularly, the approach focuses on automating the testing of both the gateway and the compatibility between IoT devices in order to reduce test cost and time to market when a new version of IoT gateway is released or the software/firmware of an IoT device is updated. Specifically, the novelty of the approach mainly consists of three parts: (1) using a CI (Continuous Integration) server along with a dedicated IoT gateway that serves as the test client to drive the tests; (2) using mock devices to simulate the connection quality of the wireless network as well as the environmental conditions; and (3) allowing testers to upload test scripts, select one or more target environments to be tested (*i.e.*, version combinations of the IoT devices), and performs system compatibility testing automatically for the devices.

To support IoT gateway testing, the approach presents an IoT gateway testing environment as shown in Fig. 4. In the testing environment, the test client and the IoT gateway to be tested (called Device Under Test, DUT) are connected to a Jenkins [16] CI server through a USB connection (to gain a faster transfer speed). The CI server serves as the master of the test client and requests the test client (*i.e.*, slave) to setup test environments, perform tests, and return test results. Upon test initialization, the corresponding target DUT firmware are downloaded from the CI server, the test client is connected to DUTs via Wi-Fi, and the DUTs are paired and connected to different physical devices such as temperature and humidity sensors through Bluetooth or USB automatically. Note that the physical devices can be replaced by mock devices (to be detailed later) for simulating different network connection qualities and device deployment environments.

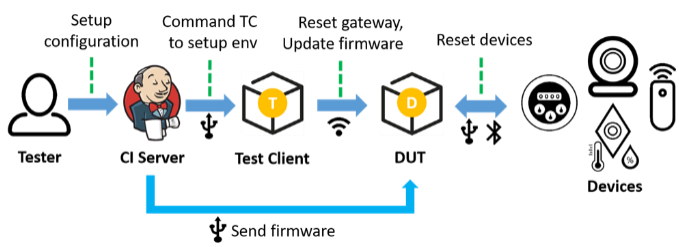

Fig. 4. The IoT gateway testing environment.

It is worth mentioning that, instead of a PC, we use the hardware of an IoT gateway to serve as the test client. This is cost effective because a gateway is significantly cheaper than a PC. The design also improves the test controllability and significantly reduces the effort and difficulty for establishing the connections between the test client and DUTs. This is because both the test client and DUT have the same hardware implementation and support the same communication protocols, such as Wi-Fi, Bluetooth, and MQTT. Thus,

the communication between the test client and DUT becomes easy. Further, since the same communication protocols are supported, the test client can also be used to connect to the physical or mock devices directly for testing the physical devices or setting the configurations of the mock devices.

3.3 The Approach for IoT Compatibility Testing

An IoT application usually consists of many types of devices that communicate with each other using different protocols. As the application and IoT technology evolve, many software/firmware versions of the IoT application and devices may be deployed at the same time. To help testers examining if an IoT application can work smoothly under different configurations (*i.e.*, target environments) and offer consistent user experience, IoT3 is designed to automate system compatibility testing, including the preparation of the target environments, test executions, test process monitoring, and test report generations.

For building the target environments automatically, IoT3 supports an IoT testing environment that consists of a cloud server and various IoT devices including mobile phone, gateway, sensors, and Wi-Fi AP. Fig. 5 shows an example of the possible 540 target environments described in Section 3.1. Specifically, the tester only needs to upload the test scripts for the IoT application under test and specify the possible versions of the IoT devices; and IoT3 will compute all possible environment configurations from the version combinations of the IoT devices. For example, a configuration can be (phone: HTC Desire, gateway firmware: 19, App version: 1.5.0, sensor version: 2.0, Wi-Fi: QA_5G). The IoT3 can then automatically execute the test scripts for each target environment sequentially and checks if the IoT application works correctly under the target environment.

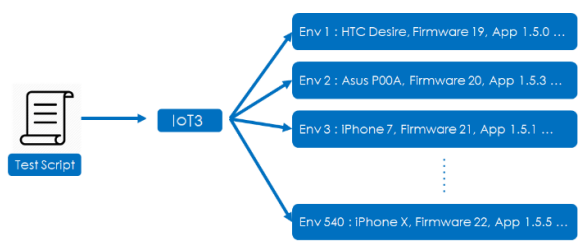

Fig. 5. An example of possible target environments for system compatibility testing using IoT3.

Since many tests need to be performed in order to verify if an IoT application is interoperable with other devices and compatible with different target environments, to provide flexibility, IoT3 allows testers to provide different test scripts as needed. In particular, to ensure interoperability and compatibility, IoT3 requires testers to provide test scripts for *App installation test* and *connection test*. IoT3 will automatically load and execute the corresponding test scripts and report the test results.

Specifically, the purpose of App installation test is to verify whether the specified version of IoT App can be installed and launched successfully on the selected target mobile phones running on special versions of operating systems. The App installation test is performed before connection test and is used to quickly identify the installation and launch

compatibility issues that can exist between different versions of the IoT App and the mobile operating systems.

The purpose of connection test is to verify that there are no communication, interoperability, and compatibility issues between different versions of the App and devices. As shown in Fig. 6, a typical connection test sequentially checks the connection or pairing between the (1) App and cloud; (2) App and gateway; (3) gateway and Wi-Fi AP; and (4) gateway and various sensor devices. For example, the connection between the App and cloud can be verified by simply testing if the App can correctly login/logout using its associated account in the cloud server. The connection between the App and gateway can be tested by adding the gateway (with specified version of firmware) to the IoT App and checking if the gateway can be added correctly. Further, the connection between gateway and Wi-Fi AP can be tested by setting the Wi-Fi AP of the gateway through the IoT App and verifying if the Wi-Fi connection can be established successfully. The connections between the gateway and various sensors can be verified by checking if the pairing of gateway with each sensor is performed successfully.

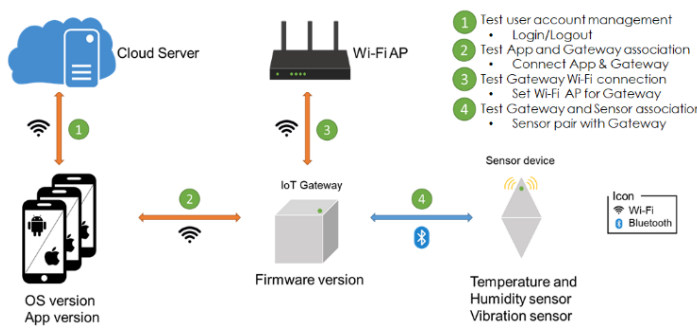

Fig. 6. The connection test of an IoT application using IoT3.

In addition to App installation test and connection test, testers can also provide additional test scripts to verify if the IoT application under test can offer some desired functionality in each specified environment configuration. Note that IoT3 currently supports the test scripts developed for Robot Framework [17], a well-known open source tool for keyword-driven testing [18]. Robot Framework enables testers to develop test scripts without the need of programming expertise; and the test scripts can also be easier to understand and maintain.

Fig. 7 shows the high-level system architecture of IoT3 system compatibility testing that consists of the Web Backend, Jenkins, Gateway Controller, IoT Environment, and Robot Framework with Appium Library [19]. The Web Backend provides the UI and functionality for testers to setup the configurations of test jobs, manage the test projects, and view the test reports. The job requests will be sent to Jenkins when test jobs for different IoT environment configurations are built and scheduled. Moreover, Jenkins will invoke the Gateway Controller to upload the corresponding firmware of gateway and setup the IoT Environment and then triggers the Robot Framework to execute the test scripts. The test results are archived in the Jenkins and can be accessed by testers through the Web

Backend. Note that, like a test client, the Gateway Controller is also made of a gateway. However, it is used for a different purpose. Thus, we name it differently.

Note that the IoT Environment in Fig. 7 consists of physical devices, including the cloud server, mobile phones, gateways, sensors, and Wi-Fi APs. Especially, for a particular version of mobile phone, sensor, or Wi-Fi AP specified in the configurations of the test jobs, the corresponding physical devices should be setup for testing. For App and firmware, the specified versions will be installed on the selected mobile phone and uploaded to the gateway at runtime automatically by IoT3. Such an approach can be beneficial in terms of test cost and flexibility since the versions of App and firmware are updated more frequently. Similarly, a physical sensor in the IoT Environment can be replaced by a mock device to increase the controllability, observability, and scalability.

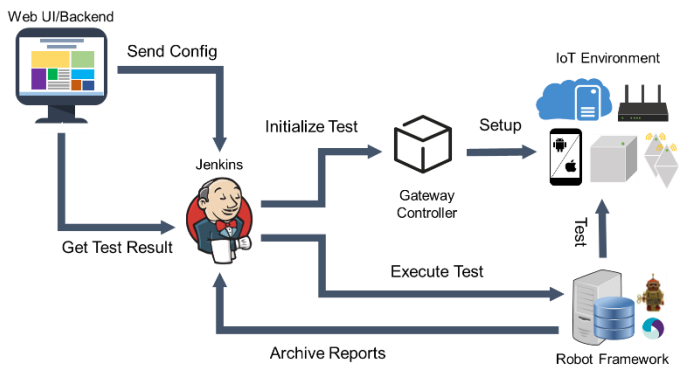

Fig. 7. The high-level system architecture of IoT3 system compatibility testing.

4. THE DESIGN AND USE OF IOT3

This section first presents the design of IoT3, including the design of test client and mock device. Then, the uses of mock devices to simulate network connection quality and environmental conditions is described. After that, running IoT compatibility tests using IoT3 is depicted.

4.1 The Design of Test Client

As described in Section 3.2, the proposed approach uses a CI server as the repository of test scripts. To perform IoT gateway testing, the test scripts are downloaded from the CI server to the test client and executed in order to verify the APIs of the DUT's firmware. Particularly, to improve test controllability and reduce test effort, the test client is made of an IoT gateway. However, an IoT gateway often has limited hardware resources, such as CPU power, memory size, and disk capacity. Thus, to allow the test scripts to be executed on the test client with restricted hardware resources, the approach uses a lightweight open-source test framework called Mocha [20], running on the Node.js [21] JavaScript runtime environment. Fig. 8 shows the major software components of the test client. The underlying operating system is Linux. The JavaScript runtime is on top of the operating system and provides the environment for running the Mocha framework and test cases.

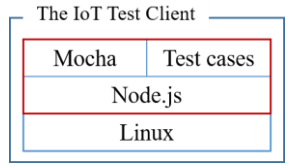

Fig. 8. The major software components of the test client.

Further, to facilitate the development and maintenance of test scripts, the test client supports keyword-driven testing method on top of the Mocha framework. Since the Mocha framework does not have built-in keyword library, we developed our own keyword library that includes a set of essential keywords specifically designed for IoT testing. Since the keywords are abstractions of script-like programs, the test cases developed using IoT3 keywords can be much easier to understand and maintain.

Fig. 9 shows an example the IoT3 keywords library and a sample test script developed using keywords. Each keyword in the library is a function which can be used to perform an action, such as turning the power switch of TV on or off. The data of the action can be passed to the keyword as parameters. In addition, testers can also create a user-defined keyword by combining a few predefined keywords. In Fig. 9, the TV test script first turns on the TV by calling the turnOnTV keyword, then switches the TV channel to 20 with the changeChannel keyword, and finally turns off the TV using the turnOffTV keyword. Note that the turnOnTV keyword uses another keyword getTVStatus with a parameter value "on" to check and assert whether the current status of the TV is indeed on. If not, the test case fails.

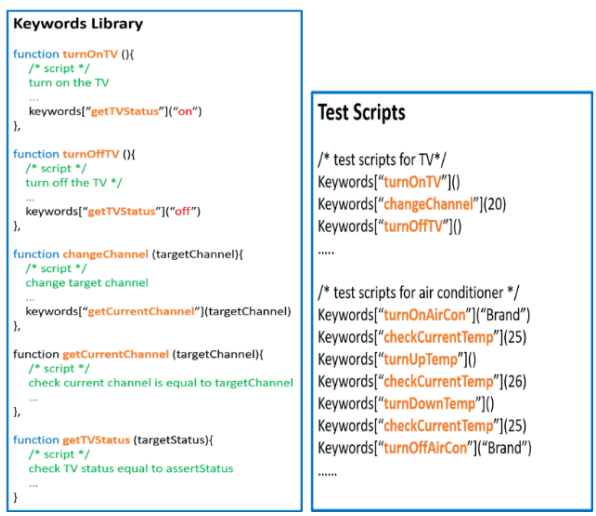

Fig. 9. An example of IoT3 keyword library and test script.

4.2 The Mock Device

As mentioned in Section 3.2, the proposed approach employs mock devices to improve testability in both IoT gateway and system compatibility testing. A mock device is

a real device with both Wi-Fi and Bluetooth adapters (*i.e.*, dongles) and configurable mock service modules. It is essentially the hardware version of a mock object [22] that is used in software unit testing. A mock device mimics the behavior of a real device, such as a temperature/humidity sensor, motion detector, *etc.* Specifically, the mock device has both Wi-Fi and Bluetooth adapters so that, like a real device, it can pair with DUT. Further, the mock device provides exactly the same API interfaces as that of a real device. Thus, from the perspectives of DUT, a mock device is not any different from a real device.

To enable a mock device to simulate the behavior of multiple types of physical devices, the API interfaces for different types of physical devices are abstracted and wrapped into a mock service module. Thus, changing the behavior of a mock device to a particular type of physical devices is as simple as selecting the corresponding mock service module. Fig. 10 shows an example. The mock device can be configured to become a selected device (*e.g.*, temperature, humidity, or motion sensor device) by the test client at the runtime. As a result, the mock device can be used to test if a DUT can communicate and co-work with the selected device.

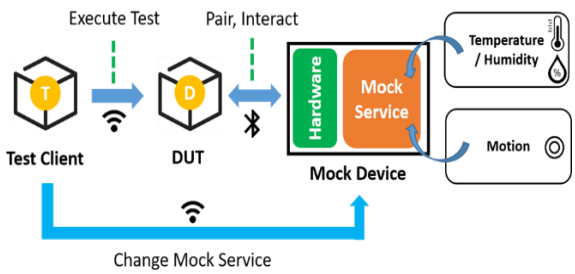

Fig. 10. An example of configuring the mock device with different services.

Fig. 11 shows how the mock devices are used to test the DUT. First, the test client sets up the mock devices by choosing the mock service module in each mock device through HTTP API requests. The test client then requests the DUT to pair and establish the Bluetooth connection between the DUT and the mock devices. Once the pairing is successful, the test client starts performing the tests for the DUT in which the DUT interacts with the mock devices according to the scenarios of the tests.

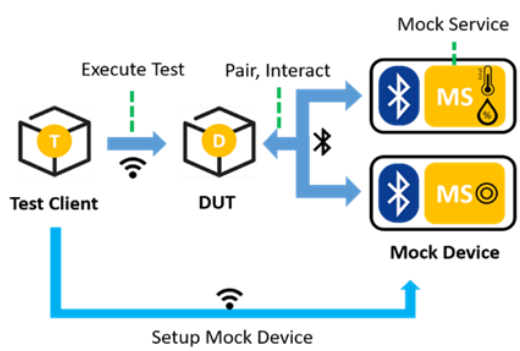

Fig. 11. The use of mock devices for unit testing DUT.

4.3 Simulating the Network Connection Quality and Environmental Conditions

To assure that the DUT can communicate and interoperate with different devices, the DUT needs to be tested under different conditions of network connection. However, the quality of wireless network connection can be affected by many factors in the surrounding environment of the DUT and devices, such as the physical distance between the DUT and the device, the obstacles blocking the signal, and the existence of external interferences. Thus, to obtain a precise control of the signal strength can be difficult.

To automate the testing of whether the DUT can withstand communication errors in an unreliable wireless network, the mock device is designed to be capable of introducing packet losses. Particularly, the Bluetooth module of the mock device can intercept and change the network packets before they are transmitted to DUT. As shown in Fig. 12, the original fragment (*i.e.*, fragment A) of a packet can be replaced by an error fragment (*i.e.*, fragment B) in order to artificially introduce the phenomena of packet losses. Further, the replacement rate can be configured to simulate the possible packet loss rate of the intended surrounding environments for the DUT and devices.

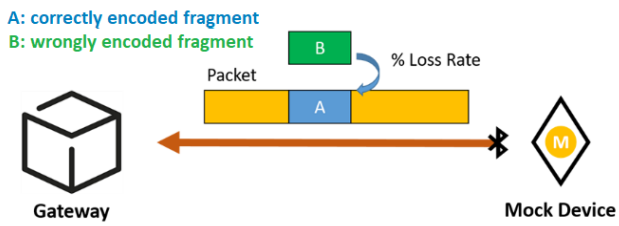

Fig. 12. An example of introducing packet losses artificially.

Moreover, to automate the tests of DUT and handle the test scenarios for the devices under different environmental conditions, especially for the extreme or harsh environments that can be difficult or expensive to replicate, the mock devices can be dynamically configured to report any assigned sensor values. This is also achieved by changing the Bluetooth packet fragments that encode the original return values of the devices with other fragments encoding the test values to be returned.

4.4 Running IoT Compatibility Tests using IoT3

As described in section 3.3, IoT3 can automatically build target IoT testing environments, execute different types of testing, and report the results of IoT system compatibility testing. The user interface is shown in Fig. 13. The testers first click "RUN TEST" link to enter "RUN TEST" page and then select the type of test (such as App installation, connection, or user defined test) as well as the versions of devices to run the compatibility test. Or, the testers can also select all types of tests and all versions of devices for testing. After that, the testers click the "Run Test" button, the corresponding test jobs will be built and executed automatically. Then, the testers can monitor the test progress and view the details of each test job, including the test time, test type, device versions, state of the test, and test report as shown in Fig. 14.

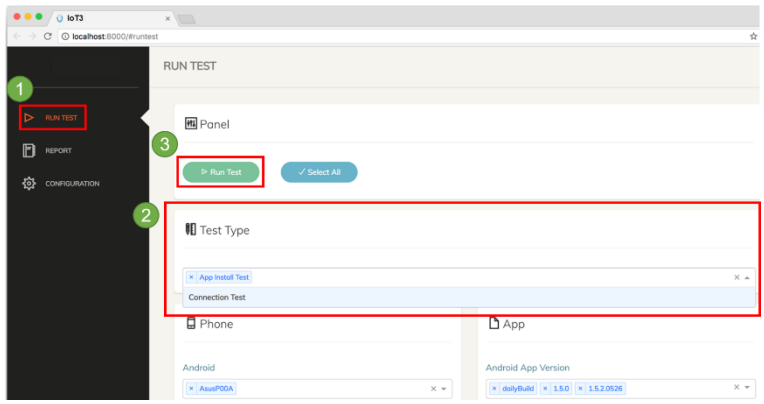

Fig. 13. A screenshot of IoT3 for selecting and running compatibility tests.

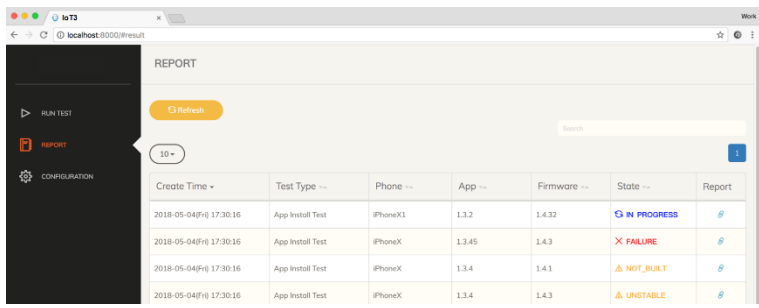

Fig. 14. A screenshot of IoT3 for reporting the compatibility test results.

5. EVALUATION

This section reports an evaluation of the proposed approach based on an industry case study. The following research questions are to be addressed.

RQ1 Can a mock device fake environmental information (*e.g.*, temperature, humidity, and motion) and send the faked information to the gateway?

RQ2 When using a mock device to simulate the packet loss of Bluetooth signal, how is the connectivity between the gateway and device under different simulated distances?

RQ3 For gateway testing, does IoT3 help to reduce testing time, human test efforts, and firmware release time?

RQ4 For gateway testing, does IoT3 help to improve test coverage and find defects that are not discovered by human testers?

RQ5 For system compatibility testing, does IoT3 offer a better target-environment coverage than human testers?

RQ6 For system compatibility testing, does IoT3 help to reduce testing time, human test efforts, and firmware release time?

The case study conducts 5 experiments on the product line of an IoT vendor, called Vendor A, that manufactures IoT devices such as the ones depicted in Fig. 1. To ensure product quality, Vendor A has a software QA department with several full-time testers. All these testers have BS or MS degrees in computer science and information engineering or in computer science and information management, and had more than 1 year of software testing experience. When a new firmware of a gateway is to be released, the firmware is manually tested by the testers using an established two-stage testing process. The two stages are S1 (*screening* testing) and S2 (*formal* testing). The screening testing takes a short time (less than an hour) that ensures no major problems exist and the new release is worthy of a full formal testing. The formal testing on the other hand is more thorough and takes more time (several hours). A new release must pass both the screening and formal testing.

After S1 and S2, a full system testing with two additional stages (S3 and S4) follows, including compatibility testing. In the third stage S3 (*quick* testing), the testers validate all the issues to be addressed have indeed been resolved. A quick testing requires about 3 working days (including the time needed for creating test reports). The last stage S4 (*full testing)* validates all system features with a lot more test cases in another 14 working days. If everything goes fine, the new firmware is released. All the four stages are performed manually by human testers without using any automated tools or scripts.

To study how the proposed approach can be applied to the products of Vendor A, we implement IoT3 to automate the IoT gateway and system compatibility testing. The test cases of IoT3 are derived from the test-case scenarios documented in S1-S4. We use Visual Studio Code to develop the mock devices and test cases, and use Jenkins to automate the scheduling of running test cases. Other devices include the Bluetooth receiver of the mock device and a Wi-Fi access point. Table 1 shows the hardware/software equipment used in the experiments.

Table 1. The experimental equipment.

Hardware/Software	Specification/Version
CPU	Intel Core i7
Memory	16G
Disk	SSD 256GB + HDD 1TB
Wi-Fi AP	TP-LINK Archer C7 AC1750
BLE Adapter	Pluggable USB Bluetooth 4.0 Low Energy Micro Adapter
OS	Ubuntu 16.04 x64
Visual Studio Code	Version 1.23
Jenkins	Jenkins ver. 2.89.2

5.1 Experiment 1

This experiment addresses RQ1. We first used a real device (a temperature sensor device) to collect all data formats and the response time of all possible interactions (APIs) between the gateway and the device. The gateway and the device were placed right next to each other so that the Bluetooth connection between them was in a perfect condition.

We then implemented a mock device by simulating the behavior of the real device based on the design described in Section 4.2. The advantage of using a mock device is that

it can be controlled by our test script to assign its sensor type and sensor information. Thus, a mock device can emulate all sorts of sensors and also allows us to fake the room environment (*e.g.*, temperature). The behavior of the mock device was tested by a test script that requested the gateway to perform pairing with the mock device and also perform all possible interactions. The test script asserted the mock device behaved exactly the same as a real device, including its response time, response data formats, and response content based on the data collected from the real device. Again, the mock device was placed right next to the gateway to ensure that their Bluetooth connection was in a perfect condition.

For a temperature sensor, there was a total of 8 different interactions and the response time varied between 10-60 seconds. In the test script, we assert each interaction with a different maximum response time and ran the test script for 100 times to ensure that the mock device behaved consistently. The results indicated that the answer to RQ1 is "yes, a mock device can fake environmental information (*e.g.*, temperature, humidity, and motion) and send the faked information to the gateway."

5.2 Experiment 2

This experiment addresses RQ2. We first measured the packet loss between the gateway and the device with different distances. We placed some other Bluetooth devices to act as interfering sources. The results are shown in Fig. 15. From a distance of 0-5 meters, the packet loss rate remained low at about 0-20%. Beyond 5 meters, depending on the source of interferences, the packet loss rate could increase dramatically from 30% to almost 100%. The packet loss rate of each distance (in meters) without any interferences is shown in Table 2.

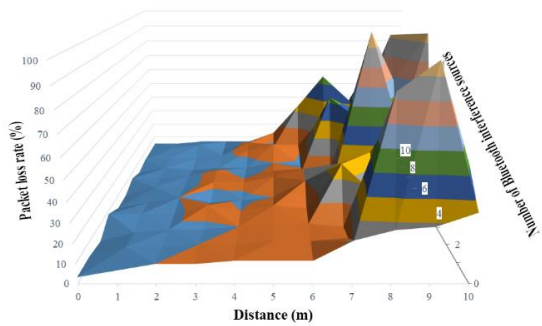

Fig. 15. The measured packet loss rate of bluetooth connections.

We then used a mock device to simulate packet loss (see Section 4.3) while the device was pairing with the gateway. A test script that requested the mock device to inject corresponding packet loss was used to perform the pairing for the distances from 0-10 meters. The test script asserted that the pairing did not exceed 60 seconds. Table 2 shows the results of running the test script for 100 times. As the distance got longer, the pairing success rate dropped. At 10 meters apart, the pairing success rate was as low as 63%. While running the test script, even though there were packet losses, the gateway under test remained intact, indicating that the design (firmware) of the gateway could withstand such packet losses without problems. Overall, the answer to RQ2 is "when using a mock device to simulate the packet loss of Bluetooth signal, the success rate of pairing the gateway and device

decreased as the distance increased. The success rate was 98% when the two devices are right next to each other."

Table 2. The pairing success rate under different simulated distances.

Distance (m)	Packet loss rate (%)	Pairing success rate
0	3.3	98%
1	6.7	93%
2	10.0	91%
3	10.0	89%
4	11.7	88%
5	11.7	89%
6	11.7	87%
7	21.7	81%
8	26.7	74%
9	28.3	69%
10	35.0	63%

5.3 Experiment 3

This experiment addresses RQ3 and RQ4. For gateway testing, we requested Vendor A to change its software testing process by using IoT3, and compared the results with Vendor A's original two-stage testing process, S1 and S2. For the measurement of the original process (or human testers), we requested two testers of the QA department to perform the screening and formal testing, and averaged their results. Both testers were highly familiar with the gateway under test.

Table 3 shows the comparison of testing time. In terms of environment setup time, it took about 6:44 (mm:ss) for IoT3 to setup the environment needed for testing. In contrast, a human tester required over 35 and 46 minutes to setup the environment for screening and formal testing, respectively. In this regard, a human tester was significantly slower than IoT3. This was because IoT3 automated all setup tasks such as resetting the gateway, arranging the network connection, and paring the gateway and device, *etc.*

Table 3. The comparison of testing time (mm:ss).

	Environment setup time	Test case execution time	Total time
S1: Manual screening testing	35:38	43:48	79:26
S2: Manual formal testing	46:50	58:11	105:01
S1 + S2: Manual testing (screening + formal)	82:28	101:59	184:27
IoT3	06:44	32:44	39:28

In terms of test-case execution time, IoT3 was also significantly faster. While a human tester took over 101 mins to carry out all test cases, IoT3 needed only 32 mins. This was not surprising since a human tester needed to manipulate the mobile App to access the gateway which was time consuming, and IoT3 did not. Overall, IoT3 was about 4.7 (184/39) times faster than a human tester. Note that the result reported in Table 3 was based

on a single round of testing. In case that a defect was found, the fixing and testing loop could be repeated for several times, and the gap between IoT3 and human testers would be widened. Consequently, once a new firmware is ready for testing, using IoT3 can significantly reduce the testing time and thus offers a faster release time.

What about human efforts? Like most automated testing approaches, it required efforts to implement IoT3. But, after IoT3 was ready, test running became automatic. No human effort was required. In contrast, 184 mins of human testing time were required with the original process. Thus, in the long run, IoT3 also saved human efforts.

Overall speaking, the answer to RQ3 is "yes, for gateway testing, IoT3 was able to help reduce testing time, human test efforts, and firmware release time."

We now turn our attention to RQ4. Table 4 shows the test coverage achieved by using IoT3 and human testers. The gateway used two different protocols, namely HTTP and MQTT. It was important to cover as many APIs as possible for both of the protocols. By following Vendor A's test cases, a human tester was able to cover 20/22 (90.9%) HTTP and 16/16 (100%) MQTT APIs, respectively, with the screening and formal testing, *i.e.*, a total of 94.7% coverage. In contrast, IoT3 was slightly better (97.4%).

Table 4. The comparison of test coverage.

	HTTP API	MQTT API	API Total	HTTP API-ISP	MQTT API-ISP	API-ISP Total
S1+S2: Manual testing	20/22 (90.9%)	16/16 (100%)	**36/38 (94.7%)**	38/62 (61.3%)	45/51 (88.2%)	**83/113 (73.5%)**
IoT3	22/22 (100%)	15/16 (93.8%)	**37/38 (97.4%)**	62/62 (100%)	50/51 (98%)	**112/113 (99.1%)**

To better identify the quality of the coverage, we use ISP (Input Space Partitioning) [23] to classify whether each input space of the APIs was covered. In this case, a human tester covered only a total of 73.5% input spaces. IoT3 on the other hand was able to cover 99.1% input spaces. Why was IoT3 better? A human tester used mobile App to perform testing which limited the kinds of APIs and the kinds of inputs that could be issued. In particular, the user interface of the mobile App blocked some illegal inputs. Thus, testing the responses of the APIs with illegal inputs became impossible. IoT3 did not have this problem. Note that as far as the quality of testing is concerned, even though illegal inputs may not occur in a real-life user scenario, they should still be tested to ensure that the gateway did not do anything wrong under such circumstances. The testing was also important to validate that the gateway conformed to its API specification so that the gateway could cooperate with any other Apps or other means of using the gateway. IoT3 did not achieve 100% coverage in MQTT API testing. This was because the current implementation of IoT3 did not have an infrared remote controller, which was required in one of the MQTT API that provided the ability of learning an infrared command from a remote controller.

The next question is whether having a higher test coverage led to finding more defects. Table 5 shows the defects detected by IoT3 and human testers at the time of the experiment. Human testers discovered 3 firmware and 3 hardware problems. In contrast, IoT3 found significantly more firmware problems. A deeper analysis revealed that the ability of testing more input spaces paid off, allowed IoT3 to detect significantly more defects. However, it should be noted that IoT3 did not detect one of the firmware defects found by a human

tester. The defect was related to turning the power of the gateway off and on again. As IoT3 did not have the ability to turn off the gateway, the defect was overlooked. Note that it is possible to equip IoT3 with a relay device (an electrically operated switch) that controls the AC power of the gateway. Having such a device allows a test case to turn the power of the gateway off and on again, and consequently allows automated detection of the aforementioned defect. We did not implement such a device in IoT3 as the automation of such testing did not appear to be an urgent matter at the time of the experiment.

Table 5. The comparison of detected defects.

	Firmware	Hardware	Total
S1+S2: Manual testing	3	3	6
IoT3	17	3	20

Overall, the answer to RQ4 is "yes, for gateway testing, IoT3 helped to improve test coverage and find defects that were not discovered by human testers."

5.4 Experiment 4

This experiment addresses RQ5. For system compatibility testing, we setup IoT3 with the following target resources: 1 cloud server, 1 Wi-Fi AP, 8 App versions (2 versions of Android Apps on 2 different Android OS versions, and 2 versions of iOS Apps on two different iOS versions), 4 gateway versions, and 6 sensor versions. The tester could request IoT3 to perform compatibility testing for up to "1×1×8×4×6=192" target environments. The test cases of IoT3 compatibility testing were derived from the test-case scenarios documented in Vendor A's system testing process S3 (quick testing) and S4 (full testing), following the approach described in Section 3.3. Note that IoT3 did not implement all test cases in S3 and S4. Instead, the IoT3 compatibility testing focused only on test cases that were related to system compatibility. In other words, IoT3 was a part of S3 and S4 and did not replace them.

We used human testers performing S3 and S4 as the reference for comparison. By studying the test cases of the quick and full testing, we identified the coverage of the target environments that was achieved. The results are shown in Table 6. Both the quick and full testing covered 16 target environments. Despite having a much longer testing time, the full testing in fact did not cover any more target environments. This was because there were a lot of test cases to perform and the compatibility testing was merely a part of the testing. Considering the release schedule and the human resource that could be allocated, the test cases selectively covered only 16 target environments as a minimal set of coverage. In comparison, IoT3 could cover 16 times of target environments with the aforementioned resources. Note that it is desirable to cover even more target environments. As long as more hardware resources are allocated, IoT3 can cover more target environments automatically. This is the strength of IoT3 over manual testing.

Overall, the answer to RQ5 is "yes, for system compatibility testing, IoT3 offered a significantly better target-environment coverage than human testers. The current IoT3 setup covered 16 times of target environments over manual testing."

Table 6. The number of target environments.

	Cloud server	Wi-Fi AP	App versions	Gateway versions	Sensor versions	Target environments
S3: manual (quick testing)	1	1	4	2	2	16
S4: manual (full testing)	1	1	4	2	2	16
IoT3	1	1	8	4	6	192

5.5 Experiment 5

This experiment addresses RQ6. Since IoT3 automates compatibility testing, all the tester needs to do is to select the target environment (or select all environments) for testing and press the execute button. It took about 5 seconds on average to do so with the Web interface of IoT3. Thus, the testing cost, the human effort spent in testing, is a small constant no matter how large the number of target environments is.

What if the compatibility testing is performed manually? We invited 4 full-time testers of Vender A to participate the experiment. We requested the testers to perform the testing for one target environment and recorded the time the testers spent. The results are shown in Table 7. On average, it took 10 minutes and 27 seconds to perform the compatibility testing for one target environment. In general, the required human effort is linear to the number of target environments to be tested. In case that there are 16 and 192 target environments to test, 2:47:16 and 33:27:12 will be required respectively (the time is estimated by multiplying the time of executing one environment with the number of environments to test). From Table 7, IoT3 could reduce the testing cost significantly, and the saving would become even more prominent if new versions are released frequently.

Table 7. The human effort required for compatibility testing.

Target environments	S3+S4: Manual (hh:mm:ss)	IoT3 (hh:mm:ss)
1	00:10:27	00:00:05
16	02:47:16	00:00:05
192	33:27:12	00:00:05

What about testing time, the time needed to complete a testing? On average, IoT3 needed 11 minutes and 32 seconds to complete the testing for a target environment, roughly 10% slower than human testers (10:27). This was because while changing the firmware version of the gateway, the current implementation of IoT3 used a fixed waiting time, which was somewhat slower than human testers who could quickly continue to the next testing step by watching the power light of the gateway. Table 8 shows the testing time required for 1, 16, and 192 target environments. Note that, despite IoT3 was slightly slower, this was not a problem at all. While IoT3 was running, the tester could do something else. In contrast, it would be unlikely for a human tester to continue working for over 8 hours without taking a rest. In other words, the time needed to wait for human testers to complete testing should be in general much longer than the testing time shown in Table 8.

But, could IoT3 discover compatibility problems? We installed IoT3 in the continuous integration system of Vendor A. From March to June 2018, IoT3 automatically dis-

covered 7 compatibility problems. On the other hand, human testers found only 3 of the 7 problems. Thus, from the viewpoint of doing compatibility testing, IoT3 was better in discovering problems.

Overall, the answer to RQ6 is "yes, for system compatibility testing, IoT3 helped to reduce testing time, human test efforts, and firmware release time?"

Table 8. The testing time required for compatibility testing.

Target environments	Manual (hh:mm:ss)	IoT3 (hh:mm:ss)
1	00:10:27	00:11:32
16	02:47:16	03:04:27
192	33:27:12	36:53:20

6. CONCLUSIONS

This paper presented a novel approach and a supporting tool called IoT3 to automate the testing of both the IoT gateways and the compatibility between IoT devices. Particularly, for gateway testing, the proposed approach employs a CI server along with a gateway as the test client to drive the testing. This enables the improvement of test controllability and reduce the effort and difficulty to establish the connections between the test client and DUTs. In addition, the test client supports keyword-driven testing method that can facilitate the development and maintenance of test scripts. Further, a mock device method is introduced to simulate the network connection quality and the environmental conditions of the devices for improving testability and facilitating IoT test automation.

For system compatibility testing, IoT3 allows testers to upload test scripts, select test environments, and perform tests to verify if the IoT application is functionality compatible with and run well on different target environments. The experimental results of an industry case study suggest that the proposed approach and the IoT3 tool can significantly save the testing time and human efforts as well as reduce the release time of new versions or software/firmware updates of IoT gateways and devices. Further, as compared with manual testing, IoT3 can also detect more faults and increase test coverage considerably. In the future, we plan to extend IoT3 to support the testing of data interoperability between devices and IoT applications. Moreover, we also plan to extend IoT3 to support end-to-end IoT testing to automatically ensure that the mobile App, gateways, devices, and cloud services together can deliver the expected functionality of IoT applications.

REFERENCES

1. IDC, "The growth in connected IoT devices is expected to generate 79.4ZB of data in 2025," https://www.idc.com/getdoc.jsp?containerId=prUS45213219, 2020.
2. A. Taivalsaari and T. Mikkonen, "A roadmap to the programmable world: Software challenges in the IoT era," *IEEE Software*, Vol. 34, 2017, pp. 72-80.
3. T. Kanstrén, J. Mäkelä, and P. Karhula, "Architectures and experiences in testing IoT communications," in *Proceedings of IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2018, pp. 98-103.

4. J. P. Dias, F. Couto, A. C. R. Paiva, and H. S. Ferreira, "A brief overview of existing tools for testing the Internet-of-Things," in *Proceedings of IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2018, pp. 104-109.
5. M. Bures, T. Cerný, and B. S. Ahmed, "Internet of Things: Current challenges in the quality assurance and testing methods," in *Proceedings of the 9th iCatse Conference on Information Science and Applications*, LNEE Vol. 514, 2018, pp. 625-634.
6. J. Hwang, A. Aziz, N. Sung, A. Ahmad, F. Le Gall, and J. Song, "AUTOCON-IoT: Automated and scalable online conformance testing for IoT applications," *IEEE Access*, Vol. 8, 2020, pp. 43111-43121.
7. S. Brady, A. Hava, P. Perry, J. Murphy, D. Magoni, and A. O. Portillo-Dominguez, "Towards an emulated IoT test environment for anomaly detection using NEMU," in *Proceedings of Global Internet of Things Summit*, 2017, pp. 1-6.
8. P. Rosenkranz, M. Wählisch, E. Baccelli, and L. Ortmann, "A distributed test system architecture for open-source IoT software," in *Proceedings of Workshop on IoT Challenges in Mobile and Industrial Systems*, 2015, pp. 43-48.
9. H. Kim, A. Ahmad, J. Hwang, H. Baqa, F. L. Gall, M. A. R. Ortega, and J. Song, "IoT-TaaS: Towards a prospective IoT testing framework," *IEEE Access*, Vol. 6, 2018, pp. 15480-15493.
10. M. R. Palattella, F. Sismondi, T. Chang, L. Baron, M. Vučinić, P. Modernell, X. Vilajosana, and T. Watteyne, ''F-Interop platform and tools: Validating IoT implementations faster,'' in *Proceedings of International Conference on Ad-Hoc Networks and Wireless*, LNCS Vol. 11104, 2018, pp. 332-343.
11. F-Interop, https://www.f-interop.eu/, 2020.
12. S. Ziegler and E. E. Kim, "Towards an open framework of online interoperability and performance tests for the Internet of Things," in *Proceedings of Global Internet of Things Summit*, 2017, pp. 1-6.
13. P. M. Pontes, B. Lima, and J. P. Faria, "Izinto: A pattern-based IoT testing framework," in *Companion Proceedings for ISSTA/ECOOP Workshops*, 2018, pp. 125-131.
14. S. T. Demirel, M. Demirel, I. Dogru, and R. Das, "InterOpT: A new testing platform based on oneM2M standards for IoT systems," in *Proceedings of International Symposium on Networks, Computers and Communications*, 2019, pp. 1-6.
15. W. Chen, C. Liu, W. W. Liang, and M. Tsai, "ICAT: An IoT device compatibility testing tool," in *Proceedings of the 25th Asia-Pacific Software Engineering Conference*, 2018, pp. 668-672.
16. Jenkins, https://jenkins.io/, 2020.
17. Robot Framework, http://robotframework.org/, 2020.
18. Keyword-driven testing, https://en.wikipedia.org/wiki/Keyword-driven_testing, 2020.
19. Robotframework-appiumlibrary, https://github.com/serhatbolsu/robotframework-appiumlibrary, 2020.
20. Mocha.js, https://mochajs.org/, 2020.
21. Node.js, https://nodejs.org/en/, 2020.
22. Mock Object, https://en.wikipedia.org/wiki/Mock_object, 2020.
23. P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, Cambridge, UK, 2008.

Chien-Hung Liu (劉建宏) received his Ph.D. degree in Computer Science and Engineering from the University of Texas at Arlington in 2002. He is currently an Associate Professor of Computer Science and Information Engineering Department at National Taipei University of Technology, Taiwan. His research interests include software testing, software engineering, software maintenance and evolution, and deep learning applications for software engineering.

Wen-Yew Liang (梁文耀) received his Ph.D. degree in Computer Science and Information Engineering from National Taiwan University in 1998. He was previously the CTO of NextDrive Corporation and is currently the Technical Director of ADLINK Technology Inc. His research interests include operating systems, computer architecture, embedded systems, edge computing, parallel and distributed systems, and low power software design.

Ming-Yi Tsai (蔡名億) received his M.S. degree in Computer Science and Information Engineering from National Taipei University of Technology in 2018. He is currently a Software Engineer of NextDrive Corporation. His research interests include IoT applications, software engineering, and software testing.

Wei-Che Chang (張偉哲) received his M.S. degree in Computer Science and Information Engineering from National Taipei University of Technology in 2018. He is currently a Software Engineer of Memopresso Inc. His research interests include IoT applications, software engineering, and software testing.

Woei-Kae Chen (陳偉凱) received M.S. and Ph.D. degrees in Computer Engineering from North Carolina State University in 1988 and 1991, respectively. He is currently a Professor of Computer Science and Information Engineering Department at National Taipei University of Technology, Taiwan. His research interests include software testing, software engineering, visual programming, and deep learning applications for software engineering.