Towards an Acceptance Testing Approach for Internet of Things Systems

Maurizio Leotta, Filippo Ricca, Diego Clerissi, Davide Ancona, Giorgio Delzanno, Marina Ribaudo, Luca Franceschini

Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS)
Università di Genova, Italy

{name.surname}@unige.it

Abstract. Internet of Things (IoT) applications and systems pervade our life increasingly and assuring their quality is of paramount importance. Unfortunately, few proposals for testing these complex — and often safety-critical — systems are present in the literature and testers are left alone to build their test cases.
This paper is a first step towards acceptance testing of an IoT system that relies on a smartphone as principal way of interaction between the user and a complex system composed by local sensors/actuators and a remote cloud-based system. A simplified mobile health (m-health) IoT system for diabetic patients is used as an example to explain the proposed approach.

1 Introduction

Internet of Things (IoT) is a network of interconnected physical objects and devices that share data through secure infrastructures and transmit to a central control server in the cloud. As the IoT technology continues to mature, we will see more and more novel IoT applications and systems emerging in different contexts. For example, trains able to dynamically compute and report arrival times to waiting passengers, cars able to avoid traffic-jam by proposing alternative paths and m-health systems able to determine the right medicament dose for a patient.

Ensuring that IoT applications are secure, reliable, and compliant is of paramount importance since IoT systems are often safety-critical. At the same time, testing these kinds of systems can be difficult due to the wide set of disparate technologies used to build IoT systems (hardware and software) and the added complexity that comes with Big Data (the three "V", huge volume, great velocity and big variety).

To the best of our knowledge, IoT software testing has been mostly overlooked so far, both by research and industry [13]. This is evident if one looks at the related scientific literature, where proposals and approaches in this context are extremely rare.

In this work, we propose a novel approach for acceptance testing of IoT systems using a UI (in our specific case study, a smartphone) as principal way of interaction between the user and the system. The approach is performed at the system level and is based on two main ingredients: virtualization and the usage of automated visual testing tools that are able to drive the UI (i.e., the smartphone). We focus on acceptance testing, a type of black box testing based on the concept of test scenario, i.e., a sequence

of actions performed on the system interfaces, which is considered by many organizations [12][1] the most effective way to ensure the quality of a fully deployed system. In fact, assembling the system and testing it as a whole is the most logical and effective way to ensure the quality. At the same time, this task can also be quite challenging for complex IoT applications.

This paper is organized as follows: Section 2 presents an invented but realistic testing scenario based on a mobile health IoT system. Section 3 describes our proposal for acceptance testing of IoT systems, with examples and a detailed description of the used testware. Finally, Section 4 presents the related works comparing them with our proposal, followed by conclusions and future work in Section 5.

2 The Testing Scenario: A Diabetes Mobile Health IoT System

We have chosen a diabetes mobile health IoT system as testing scenario for two reasons. First, these systems are incredibly difficult to test and no consolidated approaches have been proposed for them in literature. Second, many software apps for smartphones and IoT systems are now available for diabetes and are intended to assist patients to make decisions for themselves in real time [6].

This proliferation of apps and systems is due to the fact that diabetes is a very common disease that doubles a person's risk of early death. Just to give two estimates [2]: 422 million people have diabetes worldwide (2016) and the World Health Organization (WHO) reports that diabetes resulted in 1.5 million deaths in 2012, making it the 8th leading cause of death [1]. Insulin therapy is often an important part of diabetes treatment and it is injected to the patient to keep the blood glucose level low.

2.1 The Scenario

Let's suppose the following fictitious but realistic scenario. A *Company* has completed the implementation of its `DiaMH` system after several months of work and now it has to be tested. `DiaMH` is a Diabetes Mobile Health IoT system that: (1) monitors the patient's glucose level, (2) sends alerts to the patient and the doctor when a glucose level or a pattern of glucose levels is out of a pre-specified target range and, (3) regulates insulin dosing. `DiaMH`, sketched in Figure 1, consists of the following components: a wearable glucose sensor, a wearable insulin pump, a patient's smartphone, a doctor's smartphone and a cloud-based healthcare system. Glucose sensor and insulin pump are devices (respectively, the sensor and the actuator) connected to the smartphone that is used as a "bridge" between them and the cloud-based healthcare system. Moreover, the smartphone is used by the patient to visualize the glucose tendency and give commands to the system, e.g., to accept the novel dose of insulin suggested by `DiaMH`. The cloud-based healthcare system is the core of `DiaMH`, and is able to process big data and turn it into valuable information (alerts and novel doses of insulin). The complete list of functionalities of `DiaMH` is explained in the next subsection.

[1] in [12] this kind of testing is called end-to-end testing

Fig. 1. Components and actors of `DiaMH`

A thorough testing phase is required because the *Company* would like to request FDA (US Food and Drug Administration)[2] to include `DiaMH` on its approved list. The boss of the *Company* is aware that it will be difficult because few treatment apps for any disease have been approved by the FDA to actually make decisions or treatment recommendations [6]. For this reason, she forms a team of testers and selects as project manager the best one in the *Company*. His fancy name is Jim.

The team has three months of time for: (1) selecting a testing approach, (2) producing a test plan, (3) implementing the testware, i.e., the testing infrastructure[3], and (4) testing the system. The challenge seems impossible because `DiaMH` is a complex, real-time, safety critical IoT system.

Jim knows that for convincing his boss to apply to the FDA he has to quickly realize a testware able to determine whether `DiaMH` satisfies the acceptance criteria (acceptance testing).

After having surfed the Internet and consulted the literature, Jim discovers that there are no well-documented approaches that can be used for acceptance testing purposes. He found only very general non-scientific papers concerning testing of IoT systems[4] and several proposals for testing bioinformatics software (e.g., [3]). Unfortunately, these works cannot be used for his purpose since they do not describe a specific solution for testing a complex m-health IoT system like `DiaMH`. Jim is desperate. What should he do? After a first moment of discouragement a possible idea "takes shape"[5] in his mind.

A possible way to determine whether `DiaMH` satisfies the acceptance criteria is using two ingredients: test automation tools/frameworks and mock devices (e.g., virtual or simulated ones). Test automation tools are used to execute test scripts in unattended way, report the results, and compare them with earlier test runs. Test automation tools/frameworks [8] control the execution of test scripts giving commands directly on

[2] FDA is the consumer watchdog in America's healthcare system

[3] Testware includes artifacts produced during the test process such as, test scripts, inputs, expected results, set-up and clear-up procedures, files, databases, environment, and any additional software or utilities used in testing

[4] e.g., https://devops.com/functional-testing-iot/

[5] note that since `DiaMH` is a safety critical system, also other verification techniques, not considered in this paper, should be applied (e.g., runtime verification [10], model checking [4]).

the UI (e.g., the smartphone interface) and reading actual outcomes to be compared with predicted outcomes. Mock devices are pieces of software that mimic the behaviour of real devices (e.g., the glucose sensor) in controlled ways (e.g., providing glucose profiles of different kinds of diabetics patients). The idea taking shape in Jim's mind is building a set of acceptance test scripts for the acceptance criteria that, when executed, can drive the `DiaMH` system execution giving commands on the smartphone UI, as a real patient does, and verifying the correctness of the `DiaMH` system through the UI interface.

2.2 DiaMH Functionalities, Components and Protocols

This section specifies the functionalities of `DiaMH`, clarifies the role of the components, and sketches possible protocols to be used for implementing `DiaMH`. The following description has been inspired by two previous works: a Parasoft white paper [12] and the paper by Istepanian et al. [5].

The ***glucose sensor*** (e.g., SugarBeat®[6]) is a pain-free, non-invasive, needle-free sensor, that monitors the blood for detecting the glucose level, and performs measurements in timed intervals (sampling). The intervals can be set by the patient using the ***smartphone*** with a specific command (e.g., at 15 minutes intervals). Moreover, another command performs measurements on request. The glucose sensor is wirelessly connected to an app on the smartphone. The app stores the values locally and provides a UI so that the user can monitor the glucose levels in the blood and compare them with historical data. More precisely, glucose data are displayed on the smartphone screen with symbols that represent the *direction and value of glucose*. Other data displays include graphic representations indicating the *glucose pattern* over varying periods of time. The app running on the smartphone, in addition to provide simple analyses and charts to the user, transmits glucose level data to a cloud-based healthcare system for persistent storage and additional analyses.

The ***insulin pump*** follows a programmed schedule controlled wirelessly by the smartphone. Smartphone, glucose sensor and insulin pump use the Bluetooth Low energy technology[7] to communicate among them. MQTT[8] is the protocol used in the communication between smartphones, sensors, actuators, and healthcare system.

The ***healthcare system*** running on the cloud is the core of the `DiaMH` IoT system. It stores the data of all the patients, compares them to historical data, performs advanced analyses, computes the insulin quantities that should be injected by the insulin pump and sends alerts to the doctor and patient in case of danger, i.e., when problematic patterns are identified. To perform the analyses and control the insulin pump, the healthcare system uses cognitive computing, machine learning algorithms and, in general, artificial intelligence mechanisms. Indeed, establishing the "correct" dose of insulin is incredibly complex, since the insulin requirements are affected by the individual's physiology, the type and duration of daily activity, work schedule, exercise, illness and concomitant medications [11]. The alert sent to the patient contains some data, such as a guidance

[6] http://www.nemauramedical.com/sugarbeat/
[7] https://en.wikipedia.org/wiki/Bluetooth_low_energy
[8] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html

for the next steps and changes to the insulin dose (if any) to be delivered to the insulin pump. After the patient's approval, changes to the insulin dose are transmitted to the pump and set. When a problematic pattern is identified (i.e., something that requires immediate medical attention which cannot be solved only by injecting a certain quantity of insulin), the healthcare system sends an alert directly to the user's app and to the doctor's app. The alert sent to the doctor contains the GPS coordinates of the patient and a summary of her medical conditions.

3 Acceptance Testing for Healthcare IoT systems: A Proposal

Testing the example system described in the previous section poses significant challenges, given that `DiaMH` is composed of several components (applications and devices containing logics) working together and with risk of individual fail. Moreover, further problems could derive by the integration of the components. Indeed, it is well-known that an "imperfect" integration can introduce a myriad of subtle faults.

In general, a complete test plan should include a combination of unit testing (components should be isolated and tested early), integration testing (components should be tested as a group, proceeding, e.g., bottom-up), and acceptance testing, and should be conducted at two different levels:

1. testing a ***virtualized version of `DiaMH`***, where real hardware devices are not employed. In their place, Virtual devices (e.g., a mock glucose sensor) have to be implemented and used for stimulating the applications under test. At this level the goal is testing only the software produced by the *Company*, i.e., the apps (for patients and doctors) running on the smartphones and the healthcare system running in the cloud. Thus, possible unwanted behaviours of `DiaMH` due to hardware or network problems cannot be detected in this setting.
2. testing the ***real IoT `DiaMH` system*** complete of applications and devices (i.e., glucose sensor and insulin pump). The goal here is testing the system in real conditions, i.e., under real world scenarios like communication of the application with hardware, network, and other applications

Since the system is safety-critical, both testing phases should be conducted because the former (i.e., on the virtualized system) could favour earlier implementation problems detection and could potentially reveal more faults (timings of sensors and actuators can be made shorter and thus a huge quantity of tests can be run in a short time). Moreover, since `DiaMH` is a safety critical system, other useful verification techniques should be applied (e.g., runtime verification [10], model checking [4]), but these are out of the paper's goal. In this work, we mainly focus on phase (1) because: it is the first one that a test team has to face, it can be conducted without employing real sensors and actuators that can be complex to use/set and more expensive. Indeed, imagine how could be complex to test a dangerous life threatening scenario using a real glucose sensor. Moreover, since devices used in m-health systems are usually certified (e.g., from the FDA), they are not the main reason of failure of the entire system. Finally, we focus on acceptance testing because this phase seems to be neglected more than the others in the IoT domain, since the specific peculiarities of testing an IoT system (e.g., composed by

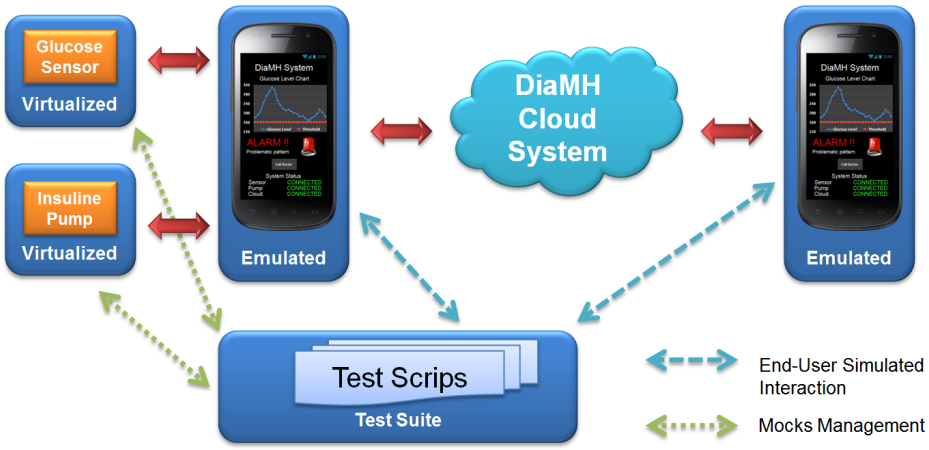

Fig. 2. Sketch of the testware used for testing `DiaMH`

different platforms, relying on various communication protocols, and including complex behaviours of smart-devices) are more evident when testing the whole system than when testing the single, isolated components.

Our proposal for facing the testing phase (1) is based on two concepts: virtualization (in particular simulation and emulation) and usage of automated visual testing tools. Simulation is used to model the devices. More precisely, the glucose sensor and the insulin pump are simulated via software, respectively, for stimulating the `DiaMH` system with selected inputs and for recovering the actual outputs of `DiaMH` to be compared with expected outputs. The real behaviour of each smartphone is emulated; that means that the real code of the smartphone will be executed on an emulator (e.g., Android Emulator[9]) able to mimic the real behaviour of the smartphone's app and visualize the UI of the smartphone on the PC screen. On the same PC an automated testing tool is installed. The testing tool will execute the test scripts that: – interact with the UI of the emulated smartphones, and – set the behaviour of the virtualized sensors (e.g., the glucose sensor should return a certain sequence of values representing a "problematic pattern"). Fig. 2 reports an overview of the elements involved in our approach: the testware (i.e., the virtualized mocks for glucose sensor and insulin pump and the test scripts), the emulated smartphones, and the healthcare cloud system.

3.1 Testware Implementation

Currently, our research group is working on the testware implementation, i.e., mocks for glucose sensor and insulin pump and the test suite for a sample system analogous, from a functional point of view, to the aforementioned `DiaMH` system.

For implementing the mocks we selected Node-RED[10], a flow-based visual programming language built on Node.js which has been expressly designed for wiring

[9] https://developer.android.com/studio/run/emulator.html
[10] https://nodered.org/

together hardware devices, APIs and online services by means of JavaScript nodes that can be easily created and combined with a rich browser-based flow editor. Moreover, Node-RED is also used to wire the mock devices with the emulated smartphones and with the healthcare cloud system using the MQTT protocol. All these software components (mock devices and emulated smartphones) will be uploaded on the IBM Bluemix[11] cloud, where the healthcare cloud system is already deployed. The execution of the emulated smartphones will allow to test the real implementation of the `DiaMH` mobile apps (i.e., one for the patients and one for the doctors). Note that since the `DiaMH` mobile apps implementation is done by relying on the Adobe PhoneGap cross-platform framework[12], iOS, Android and Windows Phone versions of the `DiaMH` mobile apps will be generated.

The second ingredient of our approach is the test suite and the corresponding testing tool. Several approaches can be employed to automate acceptance testing [8]. They can be classified using two main criteria:

1. *how test cases are developed*. It is possible to use the capture/replay or the programmable approach;
2. *how test scripts localize the UI elements to interact with*. There are three main approaches: visual (where image recognition techniques are used to locate UI elements), UI structure-based (a.k.a. DOM-based in the context of the web apps, where UI elements are located using the information contained in the Document Object Model and a locator is, for instance, an XPath expression [9]), and coordinates-based (where screen coordinates of the UI elements are used to interact with the app under test).

In our approach, we adopt the programmable approach for developing the test scripts since it provides a major flexibility over capture/replay for test development [8]. Programmable acceptance testing is based on manual creation of a test script. Test scripts can be written using ad-hoc languages and frameworks or general purpose programming languages (such as Java) with the aid of specific libraries. Usually, these libraries extend the programming language with user friendly APIs, providing commands to, e.g., click a button, fill a field and submit a form.

In this work we adopt the visual approach for interacting with the UI since test scripts have to interact with the UIs of mobile apps developed for different platforms (i.e., those supported by Adobe PhoneGap). In this case, the visual approach allows to create test scripts totally agnostic from the tested platform except, in some cases, for the locators since some UI elements are rendered differently among platforms.

Concerning the testing tool, we use Sikuli[13] that is based on the programmable and visual approaches, and developed as an open-source research project of the MIT User Interface Design Group. Sikuli is a visual technology able to automate and test graphical user interfaces using screenshot images. It provides image-based UI automation functionalities to Java programmers.

[11] https://www.ibm.com/cloud-computing/bluemix/
[12] http://phonegap.com/
[13] http://www.sikuli.org/

3.2 Test Cases, Test Scripts and Setup

In the context of this work, a *test case* is a list of actions performed on the DiaMH system followed by one or more assertions. A Sikuli *test script* is the implementation of a test case based on the Sikuli framework, and consists of a list of commands/instructions able to localize and interact with UI components, completed with JUnit assertions. Test cases must reflect the requirements of the system. To simplify the description, in the following of the paper we focus on test scripts concerning only the Android-based version of the patients mobile app. The app versions for the other platforms are functionally equivalent (unless cosmetic changes of the UIs).

We expect that the DiaMH system works correctly[14] when stimulated by the mock glucose sensor providing values taken from log files containing real glucose patterns recorded from diabetic patients (see Fig. 3). Having realistic input data is fundamental in order to test the DiaMH system under realistic conditions.

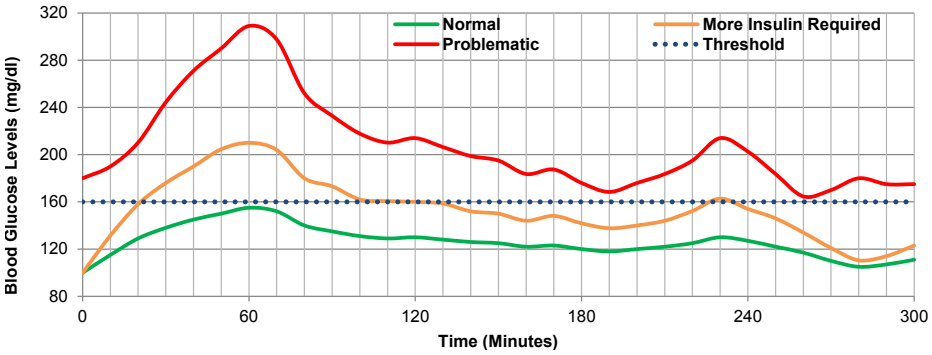

Fig. 3. Glucose patterns: Normal, More Insulin Required, and Problematic

The test suite will contain several test scripts for verifying the behaviour of the DiaMH system when different patterns are detected. For instance, when the test script that implements the test case TC1 (see below) will set the mock glucose sensor to "problematic pattern modality", we expect an alarm on the patient's smartphone. We speculate a pattern as problematic after 20 consecutive values over the threshold of 160 mg/dl of glucose in the blood (see Fig. 3).

Test case TC1

- **set** the mock glucose sensor to "problematic pattern modality" (i.e., 20 consecutive values over the threshold of 160 mg/dl)
- **set** the mock glucose sensor sampling rate at 1 read/sec [15]

[14] i.e., that it behaves as specified by the requirements in response to the currently observed glucose pattern. For instance, an alarm is displayed when a problematic pattern is detected (see Test case TC1).

[15] the log files plotted in Fig. 3 contain fictitious but realistic glucose level readings recorded every 10 minutes, thus using the mock of the glucose sensor at 1 read/sec allows to speed up the execution of the test of 600x.

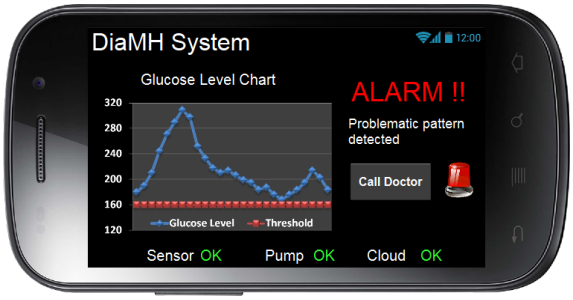

Fig. 4. Smartphone UI of the `DiaMH` app when a "problematic pattern" is detected

– after 60 seconds, **assert** alarm displayed on the patient's smartphone [16]

Fig. 4 shows what should be displayed on the smartphone screen in case a "problematic pattern" is detected by the `DiaMH` system (i.e., the interface of the `DiaMH` patient mobile app at the end of TC1, when the assertion is evaluated).

Fig.5 reports a simplified implementation of a Sikuli test script implementing TC1. The smartphone app interface is encapsulated by one (and in general more) Java classes as prescribed by the *Page Object* pattern (see DiaMH app = new DiaMH()).

```
public void testProblematicPattern(){
 commonFunction.startApp("DiaMH");
 // wait => app loading
 wait(3);
 DiaMH app = new DiaMH();
 // check if DiaMH app is started
 assertTrue(app.isTitleDisplayed(new URL("Title.png")));
 // set mock glucose sensor in "problematic pattern" modality
 GlucoseSensor gs = new GlucoseSensor();
 gs.setPatternTo("Problematic");
 // set sampling at 1 sec
 gs.setSamplingRate(1);
 // wait => pattern "Problematic" should be detected after <= 60s
 wait(60);
 assertTrue(app.isAlarmDisplayed(new URL("Alarm.png")));
}
```

DiaMH System

ALARM !!
Problematic pattern

Fig. 5. Test script for testing the display on the patient screen of an alarm raised in case of problematic pattern detection

The *Page Object*[17] pattern is a quite popular test design pattern, which aims at improving the test case maintainability and at reducing the duplication of code. A page

[16] since the mock of the glucose sensor provides a new value every second, and all of them are above the 160 mg/dl threshold, after 20 seconds the system can, in theory, raise the alarm. However, some computations are required on the cloud system. For this reason, system requirements state that the alarm must be shown on the smartphone within 40s the occurrence of a problematic pattern, thus the test verifies the presence of the alarm after 20s+40s from the first read.

[17] http://martinfowler.com/bliki/PageObject.html

object is a class that represents the UI elements as a series of class attributes and that encapsulates the features of the UI page into class methods. Adopting the *Page Object* pattern in test scripts implementation allows testers to follow the *Separation of Concerns* design principle, since the test scenario is decoupled from the implementation. Usage of *Page Object* pattern reduces the coupling between UI pages and test scripts, promoting reusability, readability and maintainability of the test suites [7] (note that Page Objects can be automatically generated [15]). Similarly, the behaviours of the glucose sensor and the insulin pump are encapsulated in classes that contain methods implementing the commands required to execute specific actions on the mocks. For instance, the method "setPatternTo" (see Fig.5) provides the mock glucose sensor with a glucose pattern recorded in an input file (i.e., a series of consecutive glucose levels).

We can see in the example two visual interactions used for evaluating the two assertions: in one case verifying that the app has been correctly loaded (hence showing the DiaMH System string, see Fig.5), while in the other verifying that the alarm message is displayed as expected. It is worth noting that Sikuli can be also used for simulating other kinds of interactions with the UI (e.g., clicking a button or typing in a field). Here, we have shown only an example of visual assertions for the sake of simplicity.

Similarly, also the other patterns can be tested by means of specific test cases. For instance, TC2 verifies the behaviour of the `DiaMH` system when the glucose levels are fine, while TC3 when an additional injection of insulin is required.

Test case TC2

- **set** the mock glucose sensor to "normal pattern modality" (slight changing values all near an optimal level of glucose and always below the threshold)
- **set** the mock glucose sensor sampling rate at 1 read/sec
- after 40, 60 and 120 seconds: **assert** NO alarm on the patient's smartphone

Test case TC3

- **set** the mock glucose sensor to "more insulin pattern modality" (in the last 20 reads there are at least 4 but not more than 15 values above the threshold of 160 mg/dl)
- **set** the mock glucose sensor sampling rate at 1 read/sec
- after 60 seconds: **assert** an increment of 2 units of insulin on the patient's smartphone

3.3 Strengths and Weaknesses of the Approach

In our opinion the major strengths of the proposed approach are the following: (1) the complexity of the entire IoT system is hidden, (2) test cases can be easily automated by adopting visual testing tools, (3) under certain conditions, test cases on a virtualized system can be executed in a fraction of the time required by the real settings.

On the other hand the weaknesses include: (1) tests are usually time consuming to develop, (2) tests are prone to non-determinism problems (i.e. the test outcomes could be influenced by factors that are outside the test control), (3) tests are prone to the fragility problem (i.e., a test that is broken when the application under test is slightly modified), (4) it is not easy to pinpoint the root cause of failure since anything in the entire flow could have contributed to the error, (5) visual test cases can verify only the functionalities of the system that provide a visible feedback on the GUI.

4 Related Work

To the best of our knowledge, this is the first paper concerning explicitly acceptance testing of IoT systems.

Rosenkranz *et al.* in [13] consider testing of IoT systems, but the goal is presenting a test system architecture for open-source IoT software instead of a general approach for acceptance testing of IoT systems as we propose. Before presenting a possible test architecture for open-source IoT systems, Rosenkranz *et al.* sketch some challenges that are specific to IoT systems. For example, heterogeneity of hardware and interoperability testing. With the Rosenkranz *et al.* proposal, we share two fundamental points: the importance of virtualization in IoT systems testing and the conviction that testing must be performed also on real devices to ensure that the combination of software and hardware works as requested.

Siboni *et al.* [14] propose a security testbed framework for testing wearable devices against external simulated attackers to cover different aspects of security design requirements. The testbed can emulate different environments (including mobile), can generate stimuli from simulated sensors, and can produce reports for data exploration and analysis. The testbed is then evaluated by a proof-of-concept. The goal of that work is different from ours: security testing for wearable devices instead of acceptance testing of a whole IoT system.

As already said, the case study used in this work has been inspired by a Parasoft white paper [12]. In that paper, the key ideas are: de-constructing the system into layers for effective testing, isolating components to improve automation and using a service testing solution for testing components. For testing the healthcare system, the service testing solution stimulates the virtualized glucose sensor and propagates to the system the obtained values. Then, the same service testing solution will be able to verify the alert irruption (if any). Our approach differs from the Parasoft's one in three respects. First, we explicitly envisage acceptance testing, while using the Parasoft's approach is difficult for a user to evaluate and accept the IoT system under test (because the interaction between the components is hidden in the service testing solution and not visible on the smartphone as in our case). Second, the solution proposed by Parasoft is more difficult to implement than ours without the commercial Parasoft tool. Third, our proposal is more complete: it tests together the smartphone's apps and the healthcare system (in particular the interactions between them), while the Parasoft proposal cannot test the software running on the smartphone (because that role is played by the Parasoft tool).

5 Conclusions and Future Work

In this work, we have presented an approach for acceptance testing of IoT systems. To explain the approach we used a realistic m-Health system composed by local sensors and actuators, a remote cloud-based healthcare system using artificial intelligence and two smartphones. However, the proposed approach is not limited to the m-health context, but can be applied to all the IoT systems that rely on smartphones as principal way of interaction between the user and the system.

As future work, we plan to complete the implementation of the testware sketched in this paper, write some test scripts and fine-tune the approach. Ongoing work is currently under-way to build up and integrate the Node-Red based mock sensors and the smartphone application. In the future, we also intend to experiment our testing approach with other IoT systems to verify its real applicability and scalability. A possible problem of the approach concerns the time for executing test scripts automatically; indeed we believe that in worst cases test suite execution could take long time.

Acknowledgements: this research was partially supported by Actelion Pharmaceuticals Italia and SEED 2016 grants.

References

1. *The top 10 causes of death*. World Health Organization (WHO), Geneva, Switzerland, 2013. http://www.who.int/mediacentre/factsheets/fs310.
2. *Global report on diabetes (1st edition)*. World Health Organization (WHO), Geneva, Switzerland, 2016. http://www.who.int/diabetes/global-report/en/.
3. T. Y. Chen, J. W. Ho, H. Liu, and X. Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics*, 10(1):24, 2009.
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, USA, 1999.
5. R. Istepanian, S. Hu, N. Philip, and A. Sungoor. The potential of internet of m-health things "m-IoT" for non-invasive glucose level sensing. In *33rd International Conference of the IEEE Engineering in Medicine and Biology Society*, EMBC 2011, pages 5264–5266, 2011.
6. D. C. Klonoff. The current status of mHealth for Diabetes: Will it be the next big thing? *Journal of Diabetes Science and Technology (JDST)*, 7(3):749–758, 2013.
7. M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proceedings of 20th Working Conference on Reverse Engineering*, WCRE 2013, pages 272–281. IEEE, 2013.
8. M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Approaches and tools for automated end-to-end web testing. *Advances in Computers*, 101:193–237, 2016.
9. M. Leotta, A. Stocco, F. Ricca, and P. Tonella. ROBULA+: An algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204, 2016.
10. M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.
11. B. H. McAdams and A. A. Rizvi. An overview of insulin pumps and glucose sensors for the generalist. *Journal of Clinical Medicine*, 5(1), 2016.
12. Parasoft. End-to-end testing for iot integrity. Technical report. https://alm.parasoft.com/end-to-end-testing-for-iot-integrity.
13. P. Rosenkranz, M. Wählisch, E. Baccelli, and L. Ortmann. A distributed test system architecture for open-source IoT software. In *Proceedings of 1st Workshop on IoT Challenges in Mobile and Industrial Systems*, IoT-Sys 2015, pages 43–48. ACM, 2015.
14. S. Siboni, A. Shabtai, N. O. Tippenhauer, J. Lee, and Y. Elovici. Advanced security testbed framework for wearable iot devices. *ACM Transactions on Internet Technology (TOIT)*, 16(4):26, 2016.
15. A. Stocco, M. Leotta, F. Ricca, and P. Tonella. APOGEN: Automatic page object generator for web testing. *Software Quality Journal*, 2016.