Towards a Runtime Verification Approach for Internet of Things Systems

Maurizio Leotta, Davide Ancona, Luca Franceschini, Dario Olianas, Marina Ribaudo, Filippo Ricca

Abstract:

Internet of Things systems are evolving at a rapid pace and their impact on our society grows every day. In this context developing IoT systems that are reliable and compliant with the requirements is of paramount importance. Unfortunately, few proposals for assuring the quality of these complex and often safety-critical systems are present in the literature. To this aim, runtime verification can be a valuable support to tackle such a complex task and to complement other software verification techniques based on static analysis and testing. This paper is a first step towards the application of runtime verification to IoT systems. In particular, we describe our approach based on a Prolog monitor, the definition of a formal specification (using trace expressions) describing the expected behaviour of the system, and the definition of appropriate input scenarios. Furthermore, we describe its application and preliminary evaluation using a simplified mobile health IoT system for the management of diabetic patients composed by sensors, actuators, Node-RED logic on the cloud, and smartphones.

Digital Object Identifier (DOI):

https://doi.org/10.1007/978-3-030-03056-8_8

Towards a Runtime Verification Approach for Internet of Things Systems

Maurizio Leotta, Davide Ancona, Luca Franceschini, Dario Olianas, Marina Ribaudo, Filippo Ricca

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

maurizio.leotta@unige.it

Abstract. Internet of Things systems are evolving at a rapid pace and their impact on our society grows every day. In this context developing IoT systems that are reliable and compliant with the requirements is of paramount importance. Unfortunately, few proposals for assuring the quality of these complex and often safety-critical systems are present in the literature. To this aim, runtime verification can be a valuable support to tackle such a complex task and to complement other software verification techniques based on static analysis and testing. This paper is a first step towards the application of runtime verification to IoT systems. In particular, we describe our approach based on a Prolog monitor, the definition of a formal specification (using trace expressions) describing the expected behaviour of the system, and the definition of appropriate input scenarios. Furthermore, we describe its application and preliminary evaluation using a simplified mobile health IoT system for the management of diabetic patients composed by sensors, actuators, Node-RED logic on the cloud, and smartphones.

1 Introduction

Internet of Things (IoT) systems are composed by interconnected physical devices that share data and often include a central remote control server on the cloud. The spread of such systems has had a significant impact on all aspects of the society and in a few years it has changed the life of billions of people. Indeed, as the IoT technology continues to mature, more and more novel IoT systems will emerge in different contexts. For example, to mention a few: smart city systems allow to intelligently provide the right level of lighting depending on human activities, time of day, season, and weather conditions hence optimizing the community energy usage relying on various kinds of interconnected sensors; mobile health systems are able to determine the optimal patient medicament doses by analysing data provided by physiological sensors [3].

Ensuring that IoT systems are secure, reliable, and compliant with the requirements is of paramount importance. Indeed, often such kind of systems are safety-critical (e.g., IoT systems used for monitoring patients, infrastructures, traffic, pollution) and given the wide set of disparate hardware and software technologies used to build them and the added complexity that comes with Big Data, this is not an easy task.

In this paper, we propose an approach based on *Runtime Verification* that can be used for assuring the quality of IoT systems. Runtime verification is a software analysis

approach in which a running system is observed by monitoring relevant events and their associated information to verify against a given (formal) specification of the expected behaviour. When efficiently implemented, the verification process can be executed (a) before the deployment of the system, for detecting bugs during development and maintenance activities, as well as, (b) after the deployment, in order to provide an additional level of protection against unforeseen events.

We will focus on detecting bugs during the development and maintenance of *Node-RED* based IoT systems. Node-RED[1] is a platform providing a flow-based visual programming language built on Node.js which has been expressly designed for wiring together hardware devices, APIs and online services. Complex behaviours can be included in the flows by using specific JavaScript nodes that can be easily created and combined in a rich browser-based flow editor [7].

This paper is organized as follows: Section 2 briefly presents a realistic scenario based on a mobile health IoT system already used in our previous complementary proposal concerning acceptance testing of IoT systems [13,15]. Section 3 describes our proposal for runtime verification of IoT systems. Section 4 reports on the empirical study evaluating our approach and the obtained results. Section 5 discusses the approach and the possible future research directions. Finally, Section 6 concludes the paper.

Related Works. To the best of our knowledge, Software Quality Assurance (SQA) of IoT systems has been scarcely investigated. Focusing on runtime verification there are no well-documented approaches except for the recent proposal of Incki et al. [9] that however is very different from ours since it adopts a different formalism for specifying the behaviour and focuses on monitoring network protocols. They used Event Calculus to specify and monitor network message exchanges in the context of Constrained Application Protocol-based IoT systems.

Also IoT software testing has been mostly overlooked so far, both by research and industry [17]. Concerning functional testing of IoT systems, excluded our recent proposal [13,15], we found only very general non-scientific papers[2] and several proposals for testing bioinformatics software (e.g., [5]). Unfortunately, these works cannot be directly used for assuring the quality of IoT systems, since they do not describe a specific testing operational procedure that a tester can follow.

2 Case Study

We have chosen a diabetes mobile health IoT system as realistic case study for two reasons. First, SQA of mobile health IoT systems is incredibly difficult, second, many IoT systems for patients management including smartphones apps are now available to assist them in making real time decisions [11].

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 the glucose level trend is out of a pre-specified target range, and (3) regulates insulin dosing. DiaMH 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

[1] https://nodered.org/ [2] e.g., https://devops.com/functional-testing-iot/

patient's smartphone (using e.g., Bluetooth LE [3]) 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 by the doctor to visualize alarms of the patients. Smartphones can access the internet by using high speed connections (e.g., UMTS[4] or LTE[5]). 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).

In this scenario a thorough SQA process is required, since DiaMH is a complex, real-time, and safety-critical IoT system.

Specification of the Required Behaviour. The goal of our proposal is to verify an IoT system, thus a precise description of its expected behaviour is required. A possible choice is to formalize it in terms of a UML State Machine (SM) to guide the SQA activities, as suggested in various model-based testing techniques, where state-based models describing systems behaviours are used, for instance, for code generation and test cases derivation [18].

Figure 1 formalizes the expected behaviour of the core part of DiaMH, i.e., the logic that recognizes the status of the patient and decides when providing an insulin dose, manages the glucose sensor and the insulin pump, and allows to show the information on the smartphones. In our simplified case study, we consider the healthcare cloud system as a deterministic system with a precise and repeatable behaviour while real systems could rely on complex algorithms based, for instance, on machine learning. Black transitions lead to DiaMH state updates while the red ones are used for managing the incoming data from the glucose sensor. Notice that each transition in the SM is composed of *trigger [guard] / actions*. The *trigger* is an optional event which activates the transition it is associated with (e.g., startApp()), the *guard* is a boolean expression which must be true to enable the associated transition (e.g., countValues(READ, THRESHOLD) ≥ 4), and the

[3] https://en.wikipedia.org/wiki/Bluetooth_Low_Energy [4] https://en.wikipedia.org/wiki/UMTS_(telecommunication)
[5] https://en.wikipedia.org/wiki/LTE_(telecommunication)

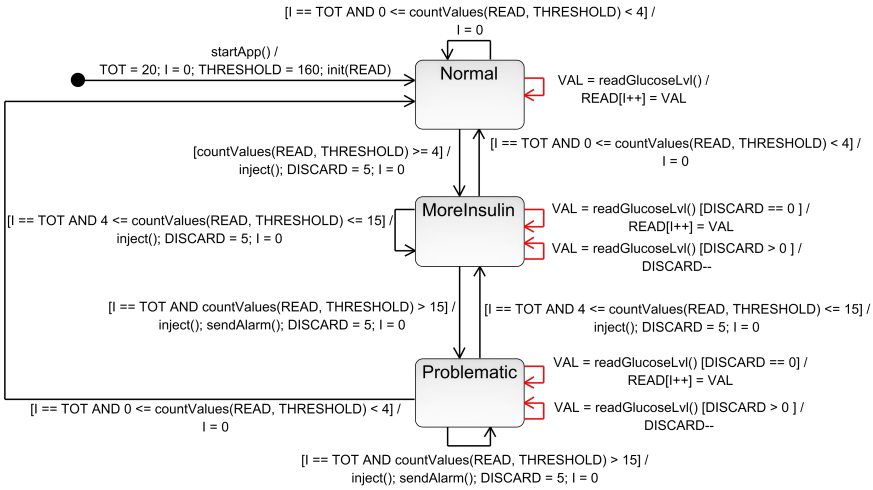

Fig. 1: DiaMH core required behaviour

actions are responses to the trigger activating the associated transition (e.g., READ[i++] = VAL).

In the SM of Figure 1, there are three states representing the possible patient's condition: *Normal*, *MoreInsulin*, and *Problematic*. When the DiaMH is bootstrapped (startApp()), the initial state is set to *Normal* and the glucose threshold (THRESHOLD) discriminating good values from bad ones is set to 160 mg/dl. In the following, we assume to set the DiaMH sampling rate to 1 Hz. Thus, each second, the glucose level is read by the sensor (readGlucoseLvl() in the red transitions) and stored in a 20 elements circular buffer (READ). If there are no more than 3 values in the buffer above the pre-set threshold the patient is considered stable and remains in the *Normal* state. Otherwise, if 4 or more values are above the threshold, a *MoreInsulin* pattern is detected. In this case, a new insulin dose is injected by the pump (inject()) and the system switches to *MoreInsulin* state, while the next 5 readings are discarded, since the injected insulin dose will need time to take effect, which we have hypothetically estimated in 5 time units (i.e., 5 seconds when the sampling rate is 1 Hz). From this state, if the next 20 readings are between 4 and 15 values above the threshold, the patient is kept in a *MoreInsulin* state, a new insulin dose is injected and the process is repeated. Otherwise, if the dangerous values are no more than 3, the patient is considered stabilized and the system goes back to the *Normal* state. Finally, in case we have more than 15 values above the threshold, a *Problematic* pattern is detected and the system moves accordingly, just after injecting a new insulin dose and notifying the problem by sending an alarm to the patient's and the doctor's smartphones (sendAlarm()). Again, from the *Problematic* state, the next 5 readings are discarded; then, after 20 further readings, the system can stay in the same state, injecting a new dose and sending a new alarm, can move back to *MoreInsulin* after an injection, or can even move to *Normal*.

3 Runtime Verification for IoT systems: a Proposal

Performing a thorough verification process for an IoT system like DiaMH is a complex task. Indeed, significant challenges have to be faced since IoT systems include several components (applications and devices containing logics) working together and with high 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 for an IoT system including software and hardware devices, SQA techniques should be conducted at two different levels:

- ***virtualized version of the IoT system***, 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 system under test. At this level the goal is verifying only the software developed, i.e., in the case of DiaMH the apps (for patients and doctors) running on the smartphones and the healthcare system running in the cloud. Thus, possible unwanted behaviours of the IoT system due to hardware or network problems cannot be detected in this setting.
- ***real IoT system*** complete of hardware devices (i.e., in the case of DiaMH glucose sensor and insulin pump). In this case the goal is verifying the system in real conditions,

i.e., under real world scenarios like communication of the application with hardware, network, and other applications.

When dealing with safety critical IoT systems, both levels should be considered and the virtualized system could favour earlier implementation problems detection, but also revealing more faults (timings of sensors and actuators can be made shorter and thus a huge quantity of scenarios can be verified in a short time). In this work, we focus on verifying virtualized IoT systems for the following reasons: it is the first step that a SQA team has to face and it can be conducted without employing real sensors and actuators that can be complex to use/set and more expensive.

Note that, beyond Runtime Verification, other two verification techniques could be applied in the case of a IoT system like DiaMH: Software Testing (e.g., acceptance testing for IoT systems [13,15] relying on E2E testing tools [14]) and Formal Verification (e.g., model checking [6]).

Figure 2 reports an overview of the elements involved in our approach: the Prolog Monitor, the Trace Expression (derived from the Behaviour Specification) provided as input to the Monitor, and the Input Scenarios sent to the IoT system. Trace Expression and Input Scenarios change for each monitored IoT system while the Prolog Monitor is general purpose and can be used for monitoring any IoT system. The monitor intercepts the monitored events by means of probes (P) intercepting events (e.g., messages) exchanged between the system components. Moreover, in the case of DiaMH we have: the mocks for glucose sensor and insulin pump, the smartphones, and the healthcare cloud system. Although these components are specific of our case study we claim that any IoT system will probably include at least a set of sensors / actuators and some distributed logic.

DiaMH virtualization Focusing on our case study, the simulation of the glucose sensor and of the insulin pump is performed via *mocks*. In this way, it is possible to provide the DiaMH system with selected input patterns and to evaluate its capability in sending

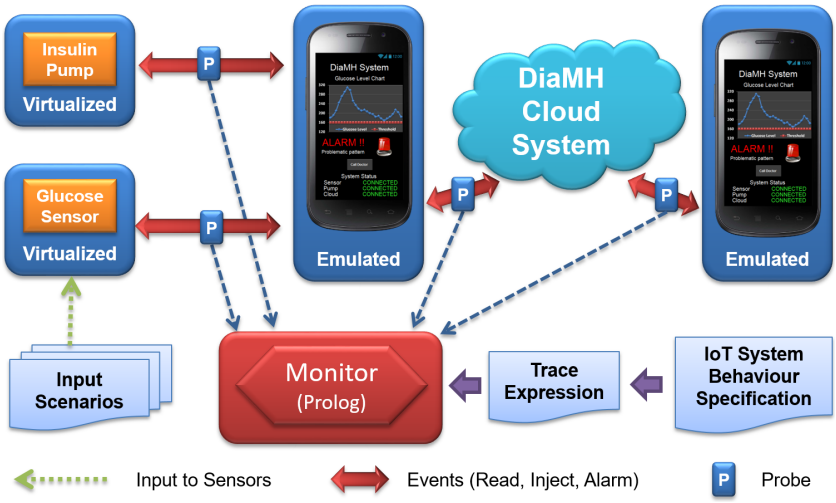

Fig. 2: Components involved in our approach instantiated for the DiaMH case study

commands to the devices (e.g., perform an insulin injection). We have chosen to implement the mocks simulating the glucose sensor and the insulin pump using Node-RED, a platform providing a flow-based visual programming language built on Node.js which has been expressly designed for wiring together hardware devices, APIs and online services. Complex behaviours can be included in the flows by using specific JavaScript nodes that can be easily created and combined in a rich browser-based flow editor [7]. Also the healthcare cloud system has been implemented in Node-RED according to the UML state machine shown in Figure 1. Instead, the emulation of the smartphones (and thus of the DiaMH UI) is based on Android Emulator[6]. Then, the logic managing the communication between the various components and the healthcare system running in the cloud has been implemented relying on Node-RED. We wired the mock devices, together with the apps running on the emulated smartphones and with the healthcare cloud system using the TCP protocol.

3.1 Monitor Implementation

Runtime Verification is a software analysis technique that aims at verifying whether a given property holds for a single run of the program under observation, rather than for any possible execution. The program is observed by a monitor, all relevant events are recorded and the resulting trace is matched against the formal specification.

The monitor has been implemented as an HTTP server able to serve the following two kinds of requests: *check-event* and *reset*.

Check-event is a POST request whose associated data must be a JSON object representing the specific event currently perceived and under monitoring; for instance, {"event":"sensor", "value":v} corresponds to a read by the glucose sensor of value v. Upon a *check-event* request, the monitor checks whether the received event is legal in the current state and responds with a corresponding JSON object {"error":b}, where b is a boolean value which is false if and only if the received event is legal; in this case, the monitor performs also a corresponding transition and updates its current state. For simplicity, the server response contains the minimum information required to conduct the experiments reported in Section 4, but a more advanced implementation of the monitor should provide a better support for error reporting.

Reset is a GET request that allows the monitor to reset its state into the initial one; this is useful for running automatically the analysis performed in Section 4 on different input scenarios.

On the client side, we have implemented a new type of Node-RED node (the Probes in Figure 2) that takes in input a JavaScript object representing an event, stringifies it in JSON, sends a *check-event* request to the monitor with the corresponding JSON object, and outputs the JavaScript object returned by the server as response.

The implementation of the monitor does not depend on the specification of the behaviour that has to be verified; such a specification must be provided separately, and fully drives the behaviour of the monitor by determining its initial state, and all possible valid state transitions in reaction of the received events. The monitor itself is implemented in SWI-Prolog[7] (around 160 SLOC). Additional details on how we implemented the SWI-Prolog monitor can be found in our previous work [2].

[6] https://developer.android.com/studio/run/emulator.html [7] http://www.swi-prolog.org

Prolog has been a natural choice as target language to support the implementation of a specification language built on top of the formalism of trace expressions; the rules of the labelled transition system defining the operational semantics of trace expressions can be directly expressed in Prolog; furthermore, the native support for cyclic terms offered by Prolog significantly simplifies the implementation of recursive trace expressions, an essential feature for runtime verification of real systems. Last, but not least, backtracking and integration with constraint logic programming are interesting additional features that can be exploited for using specifications not only for runtime verification, but also for automatically generate test cases able to improve test coverage.

3.2 Trace Expressions definition

Trace expressions [1] are a formalism explicitly devised for runtime verification which is strictly more expressive than LTL_3 [1], a temporal logic commonly used in the runtime verification context. The language features a large set of operators, including concatenation ($\tau_1 \cdot \tau_2$), union ($\tau_1 \vee \tau_2$), intersection ($\tau_1 \wedge \tau_2$) and shuffle (a.k.a. interleaving, $\tau_1 | \tau_2$); traces starting with events θ are denoted as $\theta : \tau$, while ϵ corresponds to the empty trace, i.e., no more events are expected. These basic operators can be combined to build higher-level ones, like if-then-else conditionals. Furthermore, (scoped) variables can be introduced in order to make the specification parametric w.r.t. some value that will only be known at runtime ($\{let\ x; \tau\}$). Finally, it is useful to have generic trace expressions that are parameterized: they are denoted as $\tau\langle x \rangle$.

From Behaviour Specifications to Trace Expressions. Starting from the system behaviour properly formalized by means of a SM (see Figure 1) our approach requires to define an equivalent trace expression. This task can be manually performed by an expert knowing both the SM and the trace expression formalisms. However, we believe that providing a tool for automating this step could favour the adoption of our approach among IoT practitioners (see Section 5). Moreover, note that for safety-critical IoT systems we can assume such SM-based behaviour specifications available from the early stages of design and thus even before starting the actual system development phase.

Figure 3 reports the trace expression of the DiaMH case study. The event $read(L, L', n)$ matches an input from the glucose sensor with the following constraints: L is the list of the previous values, L' is L plus the new value (if there were more than 20, the oldest one is removed), and finally there must be at least n values over the threshold. After each insulin injection ($inject$) the following 5 readings are discarded ($Discard_5$). Additionally, $read20(L, L', n)$ expects the resulting list L' to have length 20. Finally, $[\,]$ denotes the empty list.

The target language used for making the specifications executable is Prolog; hence, the formal specification of the DiaMH case study is translated into a sequence of Prolog clauses for a total of 45 SLOC, which is then fed into the monitor.

3.3 Input Scenarios definition

An Input scenario is a sequence of input values, constrained by the SM and conditions, that allows to reach a specific state following a predetermined path. Input scenarios must be carefully chosen with the goal of maximizing the coverage of the possible

behaviours of the IoT system. By analysing the SM describing the required behaviour of the system this corresponds to cover as much as possible paths among states. More specifically, for defining the paths and thus the corresponding input scenarios that lead to execute such paths, several coverage criteria could be considered (e.g., node, transition and path coverage) [4]. However, for the preliminary evaluation of our approach on the DiaMH case study, we believe sufficient to consider a limited set of input scenarios since the approach could be evaluated as even more interesting if it is able to provide good results in this setting. Thus, considering black transitions, we defined at least an input scenario that leads the system in each possible state in SM; if a state can be reached through different transitions, we chose an input scenario that reaches this state for each incoming transition. Moreover, if a state can be reached by a transition but following different combinations of states, then we chose an input scenario for each combination. Finally, for each self-loop in SM, we chose an input scenario that, before ending, follows such loop. To reduce the number of paths, we have chosen to not traverse the same transition in a single path twice (feasible in the DiaMH case study). Having this in mind we have defined ten paths covering all nodes, all transitions and an interesting subset of the possible paths (see Table 1). For instantiating the paths, actual input values are required. Note that several input scenarios can be created for each path. The careful choice of the sequence of values composing an input scenario can drastically change the effectiveness of our approach in detecting problems in the IoT system as we will discuss in Section 5. Concerning the DiaMH case study we created only an input scenario for each path by combining values from log files containing realistic glucose patterns (i.e., Normal, MoreInsulin, Problematic).

$$
\begin{aligned}
Main &= Normal\langle[]\rangle \\
Normal\langle L\rangle &= \{let\ L';\ \text{if } read(L,L',4) \text{ then } inject : Discard_5 \cdot More\langle[]\rangle \\
&\quad \text{else } read(L,L',0) : Normal\langle L'\rangle\} \\
More\langle L\rangle &= \{let\ L';\ \text{if } read20(L,L',16) \text{ then } alarm : inject : Discard_5 \cdot Problem\langle[]\rangle \\
&\quad \text{else if } read20(L,L',4) \text{ then } inject : Discard_5 \cdot More\langle[]\rangle \\
&\quad \text{else if } read20(L,L',0) \text{ then } Normal\langle L'\rangle \\
&\quad \text{else } read(L,L',0) : More\langle L'\rangle\} \\
Problem\langle L\rangle &= \{let\ L';\ \text{if } read20(L,L',16) \text{ then } alarm : inject : Discard_5 \cdot Problem\langle[]\rangle \\
&\quad \text{else if } read20(L,L',4) \text{ then } inject : Discard_5 \cdot More\langle[]\rangle \\
&\quad \text{else if } read20(L,L',0) \text{ then } Normal\langle L'\rangle \\
&\quad \text{else } read(L,L',0) : Problem\langle L'\rangle\} \\
Discard_{i>0} &= ignore : Discard_{i-1} \qquad\qquad Discard_0 = \epsilon
\end{aligned}
$$

Fig. 3: Trace expression for DiaMH

4 Preliminary Empirical Evaluation

The *goal* of our preliminary empirical evaluation is to investigate the effectiveness of the proposed runtime verification based approach. Therefore we defined the following research question:

What is the effectiveness of our approach in detecting bugs in an IoT system?
To answer the research question we used mutation analysis and the metrics used is the percentage of killed mutants out of the total.

Mutation Analysis. Mutation analysis is a technique traditionally used in the context of testing for evaluating the quality of the produced test scripts [16]. The idea is to exercise them against slight variations of the original code simulating typical errors a developer could introduce during development and maintenance activities. These variations, named mutants, can be used to identify the weaknesses in the verification artefacts by determining the parts of a software that are badly or never checked [12]. In our case the verification artefacts are the runtime monitor and its parameters: the trace expression and the input scenarios. The mutation phase is usually driven by mutation operators which affect small portions of code, exploiting some typical programming mistakes, like a change in a logical operator (e.g., AND instead of OR), a boolean substitution (e.g., from true to false), or a conditional removal (e.g., an IF condition statement is set to true). The idea is the following: monitoring each mutant to verify its correctness. The runtime monitor is effective w.r.t. a mutant if it kills the mutant, i.e., if it can detect the change in the system behaviour introduced by the mutant. Otherwise, if nothing is detected, the mutant survives and a weakness in the verification artefacts (i.e., the monitor, the trace expression, and the input scenarios) is found. The goal is to kill the highest number of generated mutants; *a measure to evaluate the overall verification artefact quality is given by the percentage of mutants killed over the total* (i.e., the higher the better).

Experimental Procedure. Starting from the implementation of DiaMH we proceeded as follows. We selected Stryker (v 0.10.3) as mutation tool, since it is a Javascript mutator, then suited for systems developed using Node-RED (v 0.17.5). Stryker supports various mutant operators, and is largely configurable to properly generate and store the mutated code. It offers mutation operators for unary, binary, logical and update instructions, boolean substitutions, conditional removals, arrays declarations, and block statements removals.

The procedure we adopted for answering the research question is the following:

- *Mutating Javascript functions*: from the original Node-RED flows implementing the core of DiaMH, by using an automated script, we selected all the function nodes embodying Javascript code and we applied Stryker on them using all the supported mutators, resulting in 29 mutants. We implemented a script that automatically and separately injects each mutated Javascript node into the original Node-RED flows, resulting in 29 mutated versions of DiaMH.
- *Mutating switch nodes*: we translated the logic embedded in the switch nodes used in the Node-RED flows in Javascript if then else statements. We mutated them with Stryker resulting in 27 mutants and applied such mutation to the original Node-RED flow, resulting in 27 mutated versions of DiaMH.
- *Evaluation*: finally, each mutated version of DiaMH (56 overall) has been executed with the ten defined input scenarios and monitored by the Prolog monitor (560 monitored executions overall), noting down: (i) whether the mutant was killed, and if so, during the execution of which input scenario and (ii) the results of a detailed analysis to explain why each mutant was killed or not.

Results. Table 1 summarizes the number of mutants killed by each input scenario. As we can see, the number of mutants killed is 44 out of 56 since 12 outlived.

Input Scenario	Transitions	Mutants Killed
from_starting_the_app_(S)_to_Normal	1	0
from_S_to_MoreInsulin	2	19
from_S_to_Problematic	3	41
from_S_to_MoreInsulin_and_back_to_Normal	3	39
from_S_to_Problematic_and_directly_to_Normal	4	43
from_S_to_Problematic_and_back_to_MoreInsulin	4	43
from_S_to_Problematic_and_back_to_Normal_(via_MoreInsulin)	5	44
from_S_to_self-loop_to_Normal	2	4
from_S_to_self-loop_to_MoreInsulin	3	38
from_S_to_self-loop_to_Problematic	4	42
Total Mutants killed - (a)		**44**
Total number of Mutants		**56**
Total number of Mutants (excluding equivalent) - (b)		**48**
Mutants detection rate - (a/b)		**92%**

Table 1: Mutants detected by our approach using the ten Input Scenarios.

We have analysed each outliving mutant from a code perspective and 8 out of 12 were identified as having an exactly equivalent behaviour with respect to the original system [8,10]. Hence there could not exist a black-box verification approach to detect them (we found them with code inspection). A simplified example is a mutation that changes `if (l==20) l=0;` to `if (l>=20) l=0;` in a Node-RED function; since the condition is evaluated for each single increment of `l`, the behaviour of the mutant is equivalent to the original code. Thus, only 4 mutants were considered as real survivors. From our analysis, we discovered that this is due to weaknesses in the provided input, since the input scenarios are not complete enough to cover all possible conditions and properly exercise the boundaries of the original system. Mutations often affect operators used in conditions. If the mutation drastically changes the behaviour (e.g., > in <) the mutant can be easily detected but, if the change is just a little variation of the system behaviour (e.g., > in >=160), the input scenario must be carefully chosen in order to detect the inconsistency. To test our conjecture, we manually created an ad-hoc input scenario for each of the 4 mutants survived. In this way, all the mutants were identified. As expected, from the results it emerges that longer input scenarios triggering several states changes in the monitored IoT system allow to detect a higher number of bugs. In the next section, we will discuss the obtained results to highlight the strengths and the weakness of the proposed approach and possible future research directions for making it applicable to real complex IoT systems.

5 Strengths/Weakness of the Approach and Future Work

Employing runtime verification for assuring the quality of IoT systems proved to be worth of investigation. Indeed, it was able to detect a relevant portion (i.e., 92%) of the bugs injected in the system implementation. This is a promising result considering that the

adopted input scenarios are largely suboptimal. From our preliminary experimentation the main weakness concerns two aspects of the input scenarios: (1) the definition of the paths to follow among the possible system states and (2) the choice of the sequences of input values useful to follow each path. We plan to replace the ad-hoc strategy used for manually defining a set of *interesting* paths by investigating how constraint solvers can be used for the generation of all the paths of bounded length. Once paths are defined we plan to investigate how effective input scenarios can be generated. Indeed, for each path several input scenarios must be created following the boundary value analysis approach extended to input sequences and taking into account the specification provided by the SM. For example, in the case of DiaMH several boundaries are present in the specification: the glucose threshold (160), the number of readings above the glucose threshold (4,15). Another aspect that could limit the adoption of our approach is the manual definition of the trace expression starting from the SM specification. For this reason, we plan to investigate the automatic generation of the trace expressions from the SM; this will allow developers without a specific background in runtime verification to adopt our method.
We plan to compare the effectiveness of runtime verification against testing (we proposed a companion approach using acceptance testing tools in [13,15]). We believe that the two approaches have different strengths and weaknesses and thus can be used together for detecting a higher number of bugs. For example, acceptance tests allow to test also the system GUI (e.g., the smartphone app) and interact with it to send commands (e.g., accepting an insulin dose if required). On the other hand, it is difficult to adopt functional test automation for automating a relevant number of test scenarios since it is by far more complex to generate test scripts involving GUI interactions. Finally, we noticed that it is not easy to pinpoint the root cause of failure when the monitor detects a problem since anything in the entire flow could have contributed. Thus, we plan to improve the monitor in order to provide detailed information concerning the current expected IoT system status when a problem is detected (currently only the corresponding value of the input scenario is reported, e.g., error @ insulin = 175).

6 Conclusions

In this work, we have presented our preliminary approach for the runtime verification of IoT systems. To explain and validate it we employed a realistic mobile health system composed by local sensors and actuators, a remote cloud-based healthcare system for taking decisions and smartphones. It is worth noting that the proposed approach is not limited to the mobile health context, but can be applied to any IoT system, providing that the messages among the devices can be intercepted by the probes of our runtime monitor. Our approach has been proposed for detecting bugs in IoT systems during development and maintenance activities; however, by removing input scenarios and relying on input provided by real sensors it can be adopted also for monitoring deployed IoT systems to provide an additional level of protection against unforeseen events. We have discussed the pro and cons of the approach and some possible interesting future research directions such as: automatically generating the input paths and input sequences to improve coverage and thus bugs detection, automatically generating the trace expressions from the system behaviour specifications, combining runtime verification with acceptance testing to

further increase bugs detection capabilities, and including time constraints in trace expressions.

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

References

1. D. Ancona, A. Ferrando, and V. Mascardi. Comparing trace expressions and linear temporal logic for runtime verification. In E. Ábrahám, M. Bonsangue, and E. B. Johnsen, editors, *Theory and Practice of Formal Methods*, volume 9660 of *LNCS*, pages 47–64. Springer, 2016.
2. D. Ancona, L. Franceschini, G. Delzanno, M. Leotta, M. Ribaudo, and F. Ricca. Towards runtime monitoring of Node.js and its application to the Internet of Things. In D. Pianini and G. Salvaneschi, editors, *Proceedings of 1st Workshop on Architectures, Languages and Paradigms for IoT (ALP4IoT 2017)*, volume 264 of *EPTCS*, pages 27–42. arXiv, 2018.
3. L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010.
4. B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., 1990.
5. 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.
6. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, USA, 1999.
7. G. Desolda, C. Ardito, and M. Matera. Empowering end users to customize their smart environments: Model, composition paradigms, and domain-specific tools. *ACM Trans. Comput.-Hum. Interact.*, 24(2):12:1–12:52, Apr. 2017.
8. B. J. Grün, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *Proceedings of 2nd International Conference on Software Testing, Verification and Validation Workshops*, ICSTW 2009, pages 192–199. IEEE, 2009.
9. K. Incki and I. Ari. A novel runtime verification solution for IoT systems. *IEEE Access*, 6:13501–13512, 2018.
10. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.
11. D. C. Klonoff. The current status of mHealth for Diabetes: Will it be the next big thing? *Journal of Diabetes Science and Technology*, 7(3):749–758, 2013.
12. P. S. Kochhar, F. Thung, and D. Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *Proceedings of 22nd Int. Conference on Software Analysis, Evolution and Reengineering*, SANER 2015, pages 560–564. IEEE, 2015.
13. M. Leotta, D. Clerissi, D. Olianas, F. Ricca, D. Ancona, G. Delzanno, L. Franceschini, and M. Ribaudo. An acceptance testing approach for Internet of Things systems. *IET Software*, 2018.
14. 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.
15. M. Leotta, F. Ricca, D. Clerissi, D. Ancona, G. Delzanno, M. Ribaudo, and L. Franceschini. Towards an acceptance testing approach for Internet of Things systems. In I. Garrigos and M. Wimmer, editors, *Proceedings of 1st International Workshop on Engineering the Web of Things (EnWoT 2017)*, volume 10544 of *LNCS*, pages 125–138. Springer, 2018.
16. A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.
17. 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.
18. M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.