Comparing Testing and Runtime Verification of IoT Systems: A Preliminary Evaluation based on a Case Study

Maurizio Leotta [a], Diego Clerissi, Luca Franceschini, Dario Olianas, Davide Ancona, Filippo Ricca and Marina Ribaudo

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

Keywords: IoT Systems, Systems Modelling, UML State Machine, Acceptance Testing, Runtime Verification, Node-RED.

Abstract: Assuring the quality of Internet of Things (IoT) systems is of paramount importance, and guaranteeing their reliability and compliance with the requirements is mandatory, but few attempts have been made so far. In previous works, we proposed two approaches for acceptance testing and runtime verification of IoT systems. Both works rely on a UML state machine to specify the system expected behaviour. In the acceptance testing approach, the interesting paths to exercise are identified and translated into executable test scripts. In the runtime verification approach, the relevant events during the system execution are monitored and compared against a formal specification derived from the UML state machine. In this paper, we compare the effectiveness of our two approaches, by applying them to a mobile health IoT system for the management of diabetic patients, employing over 100 mutated versions of the original system and analysing more than 1000 different executions. Results show that both approaches are effective in different ways in detecting bugs. While the acceptance testing approach is more effective to detect the bugs affecting the user interface, the runtime verification approach tracks better the subtle deviations from the system expected behaviour, in particular those concerning network issues.

1 INTRODUCTION

Internet of Things (IoT) systems are rapidly gaining importance as they provide a variety of useful services such as: real-time healthcare systems monitoring and assisting ill patients, smart city traffic managers, weather forecasting services, energy-saving systems built for home/office environments. IoT systems are generally composed of several interconnected and heterogeneous devices cooperating together, even from different technological domains, like web and mobile, in order to complete sometimes complex and safety-critical activities.

In this setting, proposing effective strategies for assuring IoT systems quality, for instance in terms of reliability and compliance with the requirements, is a challenging task. We have recently proposed two approaches trying to make some steps towards IoT systems quality assurance, based on *Acceptance Testing* (Leotta et al., 2018c; Leotta et al., 2018b) and *Runtime Verification* (Leotta et al., 2018a) techniques, respectively. Both proposals require, as first step, to specify the expected behaviour of the IoT system by means of a UML state machine, which plays a key role for the subsequent steps.

Concerning *Acceptance Testing* of IoT systems, we proposed an approach where the testing artefacts are generated from the UML state machine and implemented using automated acceptance testing frameworks interacting with the User Interface (UI). Concerning *Runtime Verification*, we proposed an approach in which the running system is observed by monitoring the relevant events and their associated information to be verified against a formal specification of the IoT system expected behaviour.

In this work, we compare the two approaches for better understanding their strengths and weaknesses, by taking, as case study, a UI-equipped mobile health IoT system for the management of diabetic patients, composed of a sensor, an actuator, a cloud-based healthcare system, and smartphones. While in the previous papers we empirically and separately evaluated the two approaches by assessing their capability in detecting bugs in a limited portion of the case study (i.e., the logic of the cloud-based healthcare system), in this paper we extend our analysis by including unexpected behaviours due to problems (1) in the app

[a] https://orcid.org/0000-0001-5267-0602

Leotta, M., Clerissi, D., Franceschini, L., Olianas, D., Ancona, D., Ricca, F. and Ribaudo, M.
Comparing Testing and Runtime Verification of IoT Systems: A Preliminary Evaluation based on a Case Study.
DOI: 10.5220/0007745604340441
In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, pages 434-441
ISBN: 978-989-758-375-9

running on the smartphones and (2) in the network communication among the devices.

The paper is organized as follows. Section 2 presents the case study and the specification of its expected behaviour, by means of a UML state machine. Section 3 briefly describes our two proposals for acceptance testing and runtime verification of IoT systems, while all the details can be found in the original papers. Section 4 reports the empirical evaluation and the results for comparing the effectiveness of the two approaches in revealing bugs in the considered case study. Finally, Section 5 presents the related works, followed by conclusions and future work in Section 6.

2 CASE STUDY

As case study, we implemented a Diabetes Mobile Health IoT system (from now on, DiaMH), equipped with a UI (Leotta et al., 2018b). We chose this case study because performing Software Quality Assurance (SQA) tasks on healthcare systems is generally quite complex, even more when a UI is involved (Klonoff, 2013).

The DiaMH system: (1) monitors the patient's glucose level, (2) sends to the patient and to the doctor information about the patient's state and alarms in case of problematic trends, and (3) regulates insulin dosing. DiaMH is composed of a wearable glucose sensor, a wearable insulin pump, a patient's smartphone, a doctor's smartphone and a cloud-based healthcare system. The glucose sensor and the insulin pump are worn by the patient and connected to her smartphone, which is used as a "bridge" between them and the cloud-based healthcare system. On the patient's side, the smartphone is used to visualize her state (e.g., the glucose level trend), and on the doctor's side to visualize the alarms in case of problematic trends. Finally, the cloud-based healthcare system, which is the core of DiaMH, processes the data and turns it into valuable information (alarms and novel insulin doses).

Both our acceptance testing and runtime verification proposals require to specify the expected behaviour of the IoT system under validation/verification. In our works, the specification is expressed in terms of a UML State Machine (SM) to guide the SQA activities.

Figure 1 presents the DiaMH core expected behaviour, i.e., the logic that: *(i)* determines the patient's state (i.e., Normal, More Insulin, or Problematic) and decides when to provide an insulin dose, *(ii)* manages the glucose sensor and the insulin pump, and *(iii)* transmits the information to the smartphones. In our simplified case study, we have considered the cloud-based healthcare 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. The black transitions lead to updates of the patient's state, while the red ones manage the incoming data from the glucose sensor.

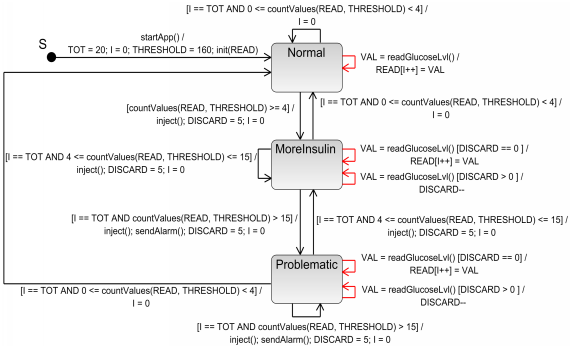

Figure 1: DiaMH core expected behaviour.

3 ACCEPTANCE TESTING AND RUNTIME VERIFICATION

Performing a thorough SQA process of an IoT system like DiaMH is complex, because it generally includes heterogeneous and interconnected components which may have to cooperate and exchange data. Faults may emerge from individual components fails or even during their integrations. Hence, SQA techniques should be conducted at two different levels: (1) *Virtual Level*, where real devices are not employed. In their place, virtual ones (e.g., a mocked glucose sensor, a smartphone emulator) are implemented and used for stimulating the system with realistic but controlled input data. At this level, the goal is to validate/verify only the software developed. (2) *Real Level*, where real devices are employed.

In this study, we focused on a virtual level, which is generally the first step that a SQA team has to face and can be conducted without employing real sensors and actuators, that are commonly complex to use/set and expensive to acquire.

We virtualized the DiaMH system by emulating the smartphones using Android Emulator[1], and by implementing in Node-RED[2] the mocks that simulate the glucose sensor and the insulin pump, the core logics of the cloud-based healthcare system and the communication among the devices. Node-RED is a platform providing a flow-based visual programming language, built on Node.js, which is expressly designed

[1] https://developer.android.com/studio/run/emulator.html

[2] https://nodered.org

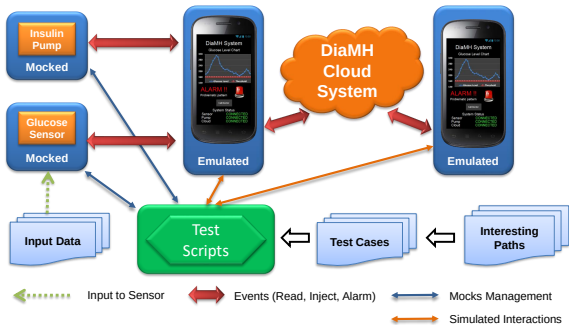

Figure 2: DiaMH Virtualization (Acceptance Testing).

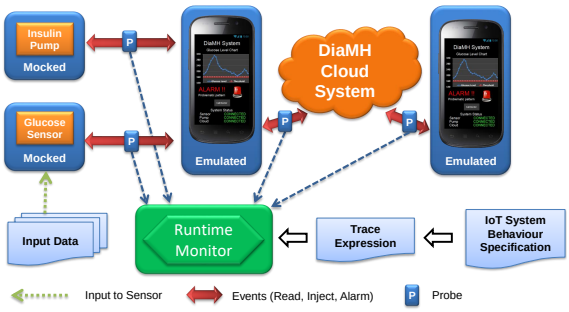

Figure 3: DiaMH Virtualization (Runtime Verification).

for wiring together hardware devices, APIs and online services.

3.1 Paths Identification

Since we focused on validating/verifying the case study from a virtual level, both our testing and runtime verification approaches require to identify from the state machine the *interesting paths* to exercise. We call a path *interesting* if it adheres to this rational: we want at least a path ending in every state of the state machine. If a state can be reached through different transitions, we want a path ending in such state for each incoming transition. Moreover, if a state can be reached by a transition following different combinations of states, we want a path for each combination. Finally, for each self-loop, we want a path presenting the self-loop as last transition.
Relying on the algorithm described in (Leotta et al., 2018b), we obtained 10 interesting paths covering all nodes and all transitions of the SM (see Table 1 under *Interesting Paths* column). For exercising these paths actual input data is required. Note that several sequences of values could be used as input data for each path, and the choice of such values could change the effectiveness of both approaches. Concerning the DiaMH case study, we limited to only one sequence of input data for each interesting path, by combining the values received from the mocked glucose sensor, which is instrumented by log files containing realistic glucose patterns to simulate the readings of the glucose level trends of different kinds of patients.

3.2 Acceptance Testing Approach

Figure 2 reports an overview of the elements involved in the acceptance testing approach: the mocked glucose sensor and insulin pump, the emulated smartphones, the test scripts implementing the test cases derived from the interesting paths (and implemented with Appium[3], a test framework for mobile apps), and the input data used to instrument the mocked glucose sensor. Refer to (Leotta et al., 2018b) for a detailed explanation of the approach.

3.3 Runtime Verification Approach

Figure 3 reports an overview of the elements involved in the runtime verification approach: the mocked glucose sensor and insulin pump, the emulated smartphones, the *trace expression* formalizing the system expected behaviour (derived from the state machine), the runtime monitor (implemented in Prolog[4]) observing all the relevant events by means of probes (P), and the input data used to instrument the mocked glucose sensor. Trace expressions are a formalism explicitly devised for runtime verification. Since we are not monitoring a real system, but a virtualized one, the relevant events occurring during the system executions have to be simulated through interesting paths fed with appropriate input data. Refer to (Leotta et al., 2018a) for a detailed explanation of the approach.

4 EMPIRICAL EVALUATION

The *goal* of the empirical evaluation is to investigate the effectiveness of our acceptance testing and runtime verification approaches in revealing bugs. The *context* of the study is defined as follows: four *human subjects* (authors of the paper) have been involved; the *software object* is the aforementioned DiaMH IoT system.

4.1 Research Questions

Our empirical study aims at answering the following two research questions **RQ1** and **RQ2**.

RQ1: *What is the capability of our acceptance testing and runtime verification approaches in detecting bugs in an IoT system?*

[3] http://www.appium.io

[4] http://www.swi-prolog.org

The goal is to quantify and compare the capability of our two approaches in revealing bugs in IoT systems; moreover, in this way, it should be possible to evaluate the differences (if any) between them. The technique adopted for evaluating the effectiveness is mutation testing (Offutt and Untch, 2001), while the metric used to answer this research question is the percentage of killed mutants out of the total. Since the location of a bug in an IoT system may affect the capability of the two approaches in revealing it, and given that IoT systems heavily rely on the network, we structured **RQ1** into three sub-research questions:
What is the capability of our acceptance testing and runtime verification approaches in:
RQ1.1: *detecting bugs in the cloud portion of an IoT system (see DiaMH Cloud System in Figures 2-3)?*
RQ1.2: *detecting bugs in the app running on the mobile devices composing an IoT system (see Emulated Smartphones in Figures 2-3)?*
RQ1.3: *detecting network problems among the devices composing an IoT system (see Red Arrows Events in Figures 2-3)?*

RQ2: *Is there a relation between interesting paths complexity and bugs detection?*
The goal is to investigate the relation between complexity of the interesting paths and their capability in detecting bugs. The complexity of an interesting path is measured in terms of the number of transitions it covers in the UML state machine, while the bugs detection capability of an interesting path is measured as the number of mutants killed during the execution of the corresponding SQA artefacts (i.e., the test script and the runtime monitor, respectively implementing the path and monitoring the events raised while using some tailored input data). To answer this research question we computed the Pearson correlation coefficient[5] between the number of transitions covered by an interesting path and the number of bugs it has detected.

4.2 Mutation Testing

Mutation testing (Offutt and Untch, 2001) is a technique traditionally used for evaluating the quality of the produced test scripts, by exercising them against slight variations of the original code simulating errors a developer could introduce during development and maintenance. These variations, named mutants, can identify the weaknesses in the test artefacts, by determining the software portions that are badly or never tested (Kochhar et al., 2015).

Generalizing to SQA, the idea is the following: for each mutant, the SQA artefacts are executed (e.g., the test scripts or the runtime monitor). A SQA artefact is effective w.r.t. a mutant if it kills the mutant, i.e., it can detect the change in the system behaviour introduced by the mutant. Otherwise, the mutant survives, proving the SQA artefact weakness in exercising such portion of the code. The goal is to kill the highest number of generated mutants; *a measure to evaluate the overall SQA artefacts quality is given by the percentage of mutants killed over the total* (i.e., the higher the better).

4.3 Experimental Procedure

Starting from the implementation of DiaMH and the two kinds of SQA artefacts created by following our approaches (i.e., the test scripts and the runtime monitor), we proceeded as follows.

To answer **RQ1.1**, we selected Stryker[6] as mutation tool, which is a Javascript mutator suited for systems developed using Node-RED, like DiaMH. Stryker supports various mutant operators and plugins, and is largely configurable to properly generate and store the mutated code. It offers mutation operators for logical instructions, boolean substitutions, conditional removals, arrays declarations, block statements removals, and so on. In particular, we focused on:

Mutating Javascript functions nodes: 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, obtaining 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[7] nodes used in the original Node-RED flows as if-then-else Javascript statements. We mutated them with Stryker, obtaining 27 mutants, and consequently, 27 corresponding mutated versions of DiaMH, automatically generated as described above.

To answer **RQ1.2**, we selected MDroid+[8] mutation tool (Linares-Vásquez et al., 2017) to mutate the Java code implementing the DiaMH app running on the smartphones, with assignments substitutions, logical operators changes and conditionals removals, resulting in 40 mutants.

To answer **RQ1.3**, we manually created 7 versions of DiaMH simulating network problems, i.e., delayed and undelivered messages. More in detail, we created: 2 versions simulating delays by 2 and 4 seconds of each network message sent from the glucose sensor to the cloud, 2 versions simulating delays by 2 and 4

[5] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

[6] https://stryker-mutator.github.io

[7] https://nodered.org/docs/user-guide/nodes#switch

[8] https://research-appendix.com/mdroid

seconds of each network message sent from the cloud to the insulin pump, and finally 3 versions simulating an undelivered message every 5, 10, and 15 messages sent from the glucose sensor to the cloud.

Finally, each mutated version of DiaMH (103 overall) was validated separately by both kinds of SQA artefacts produced with our approaches, i.e., the Appium test scripts and the Prolog monitor, using the proper input data to instrument the mocked glucose sensor, in order to purposely exercise exactly one by one the 10 interesting paths of DiaMH, resulting in 1030 overall executions (10 interesting paths for 103 mutants) for each approach, noting down: (i) whether the mutant was killed, and if so, during the execution of which interesting path, and (ii) the results of a detailed analysis to explain why each mutant was killed or not.

To answer **RQ2**, we checked the number of transitions covered in the SM of Figure 1 by each interesting path and compared it with the number of mutants each SQA artefact had killed in such case.

4.4 Results

Table 1 column **RQ1.1** *Cloud Mutants Killed* summarizes the number of mutants in the cloud killed by the Appium test scripts and by the Prolog monitor. Among the generated 56 mutants, 14 outlived acceptance testing and 12 outlived runtime verification.

We analysed each outliving mutant from a code perspective and, after some code inspection, we discovered that the behaviour of 8 of them was exactly equivalent to the original system (Grün et al., 2009; Jia and Harman, 2011), hence undetectable by any black-box SQA approach. An example of an equivalent mutation is changing if $(i == 20)$ $i = 0$ to if $(i >= 20)$ $i = 0$ in a Node-RED function; since the condition is evaluated for each single increment of i, and the initial value of i is below 20, the behaviour of the mutant is equivalent to the original code.

Thus, only 6 and 4 mutants were considered as real survivors for acceptance testing and runtime verification approaches, respectively. From our analysis, we discovered that 3 of them survived, in both our approaches, due to weaknesses in the provided input data, which was not complete enough to cover all the possible conditions and properly exercise the boundaries of the original system. Indeed, mutations often affect operators used in conditions; if the mutation drastically changes the system behaviour (e.g., $>$ in $<$), the mutant can be easily detected, but if it is just a little variation of the system behaviour (e.g., $>$ in $>=$), the input data must be carefully chosen in order to detect the inconsistency. To test our conjecture, we manually created an ad-hoc sequence of values for each of the 3 mutants that survived because of weak input data, and all of them were identified by both our approaches. Concerning the 3 other mutants surviving only the acceptance testing approach, we recognized that they were slight mutations of the DiaMH behaviour having no effect on the UI, hence impossible to be killed by the test scripts, that are highly based on assertions over UI properties. For example, a mutant changed the number of readings required to determine a patient's state from 20 to 19: while no test script has assertions to verify the exact number of received readings that bring to a change of a patient's state, the Prolog monitor looks at each message exchanged among the devices and can identify when something expected is missing. Finally, the last mutant surviving only runtime verification had caused the crash of the app and the freezing of the UI at the end of its execution, due to a continuous and erroneous increment of the size of an array. In that case, while the mutant was detected by some test scripts, since the UI had stopped displaying the proper contents after the crash, no unexpected events were detected by the monitor.

Table 1 column **RQ1.2** *App Mutants Killed* summarizes the number of mutants in the app running on the smartphones killed by the Appium test scripts and by the Prolog monitor. Among the generated 40 mutants, 15 and 33 outlived acceptance testing and runtime verification, respectively. As for RQ1.1, we analysed them and we found that 15 out of 40 were equivalent. For example, there were mutants applying trivial and undetectable UI changes (e.g., font colour/style or unchecked visual content) or initializing variables by negative or null values, which were reinitialized before their usages. Hence, excluding the equivalent mutants, in this case, the acceptance testing approach resulted by far more effective than the runtime verification (100% versus 28%), because the 18 mutants out of 33 surviving runtime verification involved changes in the UI of the app without affecting the components communication (e.g., the label describing the current patient's state was forcefully changed by a mutant independently of the exchanged readings). These mutants could not be detected by the Prolog monitor, which was able instead to detect those producing unexpected events altering the system expected behaviour, as in the case of a mutant setting to false a conditional expression used for establishing the communication.

Table 1 column **RQ1.3** *Network Mutants Killed* summarizes the number of mutants concerning network problems, i.e., delayed and undelivered messages, killed by the Appium test scripts and by the Prolog monitor. In this case, the runtime verification approach was able to detect all the mutants, while the acceptance testing approach detected only 5 out of 7. The

Table 1: Mutants killed by the Appium test scripts and by the Prolog monitor.

Interesting Paths		# Transitions	RQ1.1: Cloud Mutants Killed		RQ1.2: App Mutants Killed		RQ 1.3: Network Mutants Killed		Total Mutants Killed	
			Acceptance Testing	Runtime Verification	Acceptance Testing	Runtime Verification	Acceptance Testing	Runtime Verification	Acceptance Testing	Runtime Verification
from starting the app (S) to Normal		1	4	1	12	0	0	0	16	1
from S to More Insulin		2	10	19	14	7	0	4	24	30
from S to Problematic		3	32	41	20	7	2	7	54	55
from S to More Insulin and back to Normal		3	19	39	14	7	1	6	34	52
from S to Problematic and directly to Normal		4	41	43	20	7	2	7	63	57
from S to Problematic and back to More Insulin		4	37	43	20	7	1	7	58	57
from S to Problematic and back to Normal (via More Insulin)		5	42	44	21	7	4	7	67	58
from S to self-loop to Normal		2	4	4	12	0	0	0	16	4
from S to self-loop to More Insulin		3	19	38	14	7	1	7	34	52
from S to self-loop to Problematic		4	34	42	18	7	4	7	56	56
Total Mutants killed	**(a)**		**42**	**44**	**25**	**7**	**5**	**7**	**72**	**58**
Total number of Mutants			**56**	**56**	**40**	**40**	**7**	**7**	**103**	**103**
Total number of Mutants (excluding equivalent)	**(b)**		**48**	**48**	**25**	**25**	**7**	**7**	**80**	**80**
Mutants detection rate	**(a/b)**		**88%**	**92%**	**100%**	**28%**	**71%**	**100%**		

surviving mutants were those respectively delaying by 2 seconds the messages sent from the glucose sensor to the cloud and from the cloud to the insulin pump. Differently from the runtime verification approach, the acceptance testing approach required to introduce some temporal waits into the test scripts (e.g., to perform a refresh of the UI), which affected their actual execution time and reduced the overall precision in detecting small timing deviations from the DiaMH expected behaviour. Concerning undelivered messages, for the acceptance testing approach, only the test script exercising the longest interesting path (i.e., from S to Problematic and back to Normal (via More Insulin)) was able to kill the mutant which simulated an undelivered message every 15 messages sent from the glucose sensor to the cloud. Instead, the SQA artefacts exercising shorter interesting paths (e.g., from starting the app (S) to Normal and from S to self-loop to Normal) were not able to kill any mutant concerning network problems, neither in acceptance testing nor in runtime verification, since they used input data having very similar values and their execution ended before the mutation had occurred.

Summary: the results show the effectiveness of both acceptance testing and runtime verification approaches. Acceptance testing proves to be by far more effective than runtime verification in detecting mutants in the app affecting the UI (100% versus 28% for acceptance testing, RQ1.2). Instead, runtime verification is much more precise in tracking subtle changes in the system behaviour, in particular in the messages exchanged among the DiaMH components: indeed, it is slightly better in killing mutants in the cloud (92% versus 88% for runtime verification, RQ1.1) and those concerning network problems, where even small deviations from the expected behaviour, like delayed and undelivered messages, are detected (100% versus 71% for runtime verification, RQ1.3). If we exclude the equivalent outliving mutants (8 from the cloud and 15 from the app, see Table 1) from the total (56 from the cloud, 40 from the app, and 7 from the network, see Table 1), by combining our approaches, only 3 mutants out of 80 survive (see results of RQ1.1 concerning input data selection), hence over 96% of the generated mutants that actually modify the behaviour of DiaMH are killed. The results hint that each approach has to be chosen depending on the kind of IoT system that has to be developed and validated/verified. In many cases, both approaches may be combined to improve the overall capability in detecting bugs in IoT systems.

Table 1 also provides the information for answering to **RQ2**. It is interesting to notice that, in both our approaches, all the interesting paths involving a Problematic pattern in terms of a patient's state (i.e., at least 3 transitions of the SM) were able to kill more mutants than shorter paths (54 out of 80 in the worst case, see the row corresponding to from S to Problematic path in Table 1 for the *Acceptance Testing* approach). This can be trivially explained by Figure 1: to reach a Problematic state, it is necessary to traverse more transitions than in simpler paths; hence, whenever a Problematic pattern is taken in input, a larger portion of the system will be exercised, and, similarly, more instructions will be performed and more events will occur (e.g., inject an insulin dose, discard immediately subsequent values, send alarms, and so on), which may potentially lead to detect a large number of mutants. Two paths presented a very low mutants detection rate in both our approaches: from starting the app (S) to Normal and from S to self-loop to Normal. The low mutants detection rate for these paths is explainable by their simplicity: they verify that DiaMH is running and, after a certain amount of time, they check that the patient is still in a Normal state.

Figure 4 shows a scatter plot that displays the results concerning three variables for each interesting path: the number of traversed transitions (x-axis), the number of killed mutants (y-axis), and the adopted SQA approach (black for acceptance testing, red for runtime verification). From Figure 4, a linear relationship emerges (R^2 coefficient[9] is quite close to 1 for

[9] https://en.wikipedia.org/wiki/Coefficient_of_determination

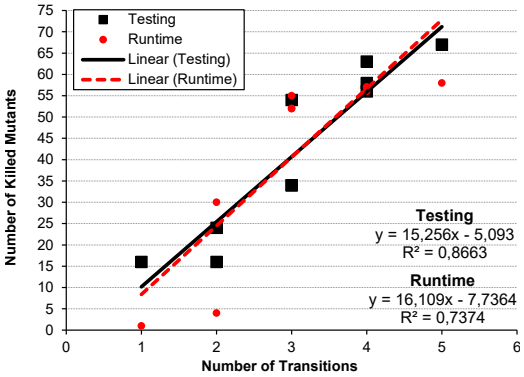

Figure 4: Paths complexities and mutants detection.

both testing and runtime verification approaches, indicating that the regression lines fit well the data) and the Pearson correlation index confirms a strong positive correlation between the number of transitions traversed and the number of mutants killed: 0.93 for acceptance testing and 0.86 for runtime verification. Indeed, the approaches present very similar regression lines; the differences are mainly based on the kinds of mutants to detect (i.e., cloud, app, and network mutants, see Table 1): in general, and for shorter paths is even more evident, the monitor is incapable to detect the mutants affecting the app, while the testing approach presents better results in the majority of paths, since is able to detect all the mutants in the app, which are almost a third of the total mutants (25 out of 80), but misses some of those affecting the communication.

Summary: to summarise, the obtained results show that high complexities in interesting paths, measured as the number of different transitions traversed in the UML state machine, correspond to high mutants detection rates (Pearson correlation coefficient close to 0.9 for both the considered SQA approaches). These results may help in choosing the right trade-off between simplicity and effectiveness of SQA artefacts and in prioritizing the executions of those paths that best answer to a project needs.

4.5 Threats to Validity

In the following, for space reasons, we sketch only some of the most relevant threats to validity of the present study.

Authors' bias threat. To limit this threat, we divided the following tasks among the authors: the system expected behaviour specification, the system virtualization and implementation with Node-RED, the acceptance testing and runtime verification approaches application to the case study, and the evaluation of the approaches by means of mutation testing, which mostly involved automated mutation tools (Stryker and MDroid+) with default settings.

Internal validity threat. To limit this threat, we adopted a systematic approach for the generation of realistic input data and for the application of the mutation technique.

External validity threat. To limit this threat, we implemented DiaMH by following some existing diabetes control systems descriptions (Parasoft, 2016; Istepanian et al., 2011), considering also other domains where UI-based components, sensors and actuators are involved.

5 RELATED WORK

For space reasons, we briefly describe only few of the scientific works existing in literature that are related to our approaches.

The acceptance testing approach has been inspired by a Parasoft white paper (Parasoft, 2016), where the components of a healthcare system are isolated and stimulated for testing purposes, verifying any alert irruption. Rosenkranz *et al.* (Rosenkranz et al., 2015) present a test system architecture for open-source IoT software. They recognize the two levels for testing IoT systems that we introduced in our works, i.e., virtual and real, but a general approach for acceptance testing is not provided. Silva et al. (Silva et al., 2014) propose a model-based architecture for generating regression models to simulate patients vital signs and emulate medical devices and actuators, by means of stored clinical data, medical guidelines and statistical techniques. A controller model is introduced to check the events and coordinate the actions among the devices and the patient models, which could also be reused and adapted depending on the clinical scenario. Siboni *et al.* (Siboni et al., 2016) propose and evaluate a testbed framework for security testing of wearable devices in terms of design requirements, where external attackers and sensors are simulated and stimuli are generated accordingly to the testing purposes.

Concerning monitoring and formal verification, Kane et al. (Kane et al., 2014) present a runtime monitor verification technique and a formal method to describe and detect violations of high-level critical properties of a safety-critical system (a vehicle simulator) and analyse resulting log data. A graphical tool is implemented by Hoxha et al. (Hoxha et al., 2014) to guide the user in formulating, visualizing, and monitoring temporal logic sentences, used to express design requirements properties of cyber-physical systems (e.g., reachability, safety).

6 CONCLUSIONS

In this work, we have compared our SQA approaches to IoT systems acceptance testing and runtime verification, in order to highlight their strengths and weaknesses while applied on realistic scenarios.

As case study, we have chosen a simplified mobile health IoT system for the management of diabetic patients, composed of a sensor, an actuator, a cloud-based healthcare system, and smartphones, having its logics partially implemented in the Node-RED environment. We have then separately applied our approaches to the case study and extended our preliminary analyses to collect data in terms of mutants detection. Results have shown that the acceptance testing approach is more effective to detect mutants affecting the UI of the app running on the smartphones, part of the system, while runtime verification proves to be more powerful in tracking subtle deviations from the system expected behaviour, in particular those affecting network issues, like delayed and undelivered messages. By combining our approaches, the mutants detection for the considered case study is over 96%.

As future research, we intend to experiment our approaches on other IoT systems to verify their real applicability and scalability, and to overcome the emerged limitations (in particular those concerning the selection of realistic input data and precise timing constraints).

ACKNOWLEDGEMENTS

This research was partially supported by Actelion Pharmaceuticals Italia.

REFERENCES

Grün, B. J., Schuler, D., and Zeller, A. (2009). The impact of equivalent mutants. In *Proc. of 2nd Int. Conf. on Software Testing, Verification and Validation Workshops*, ICSTW 2009, pages 192–199. IEEE.

Hoxha, B., Bach, H., Abbas, H., Dokhanchi, A., and Kobayashi, Y. (2014). Towards formal specification visualization for testing and monitoring of cyber-physical systems. In *Proc. of Workshop on Design and Implementation of Formal Tools and Systems*, DIFTS 2014.

Istepanian, R., Hu, S., Philip, N., and Sungoor, A. (2011). The potential of Internet of m-health Things "m-IoT" for non-invasive glucose level sensing. In *Proc. of 33rd Int. Conf. of the IEEE Engineering in Medicine and Biology Society*, EMBC 2011, pages 5264–5266.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

Kane, A., Fuhrman, T., and Koopman, P. (2014). Monitor based oracles for cyber-physical system testing: Practical experience report. In *Proc. of 44th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks*, DSN 2014, pages 148–155. IEEE.

Klonoff, D. C. (2013). The current status of mHealth for Diabetes: Will it be the next big thing? *Journal of Diabetes Science and Technology*, 7(3):749–758.

Kochhar, P. S., Thung, F., and Lo, D. (2015). Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *Proc. of 22nd Int. Conf. on Software Analysis, Evolution and Reengineering*, pages 560–564. IEEE.

Leotta, M., Ancona, D., Franceschini, L., Olianas, D., Ribaudo, M., and Ricca, F. (2018a). Towards a runtime verification approach for Internet of Things systems. In *Proc. of 2nd Int. Workshop on Engineering the Web of Things (EnWoT 2018)*, volume 11153 of *LNCS*, pages 83–96. Springer.

Leotta, M., Clerissi, D., Olianas, D., Ricca, F., Ancona, D., Delzanno, G., Franceschini, L., and Ribaudo, M. (2018b). An acceptance testing approach for Internet of Things systems. *IET Software*, 12:430–436.

Leotta, M., Ricca, F., Clerissi, D., Ancona, D., Delzanno, G., Ribaudo, M., and Franceschini, L. (2018c). Towards an acceptance testing approach for Internet of Things systems. In *Proc. of 1st Int. Workshop on Engineering the Web of Things (EnWoT 2017)*, volume 10544 of *LNCS*, pages 125–138. Springer.

Linares-Vásquez, M., Bavota, G., Tufano, M., Moran, K., Di Penta, M., Vendome, C., Bernal-Cárdenas, C., and Poshyvanyk, D. (2017). Enabling mutation testing for android apps. In *Proc. of 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 233–244. ACM.

Offutt, A. J. and Untch, R. H. (2001). Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer.

Parasoft (2016). End-to-end testing for IoT integrity. Technical report. https://alm.parasoft.com/end-to-end-testing-for-iot-integrity.

Rosenkranz, P., Wählisch, M., Baccelli, E., and Ortmann, L. (2015). A distributed test system architecture for open-source IoT software. In *Proc. of 1st Workshop on IoT Challenges in Mobile and Industrial Systems*, IoT-Sys 2015, pages 43–48. ACM.

Siboni, S., Shabtai, A., Tippenhauer, N. O., Lee, J., and Elovici, Y. (2016). Advanced security testbed framework for wearable IoT devices. *ACM Trans. Internet Tech. (TOIT)*, 16(4):26.

Silva, L. C., Perkusich, M., Bublitz, F. M., Almeida, H. O., and Perkusich, A. (2014). A model-based architecture for testing medical cyber-physical systems. In *Proc. of 29th Symposium on Applied Computing*, SAC 2014, pages 25–30. ACM.