MATTER: A tool for generating end-to-end IoT test scripts

Dario Olianas¹ · Maurizio Leotta¹ · Filippo Ricca¹

Accepted: 14 June 2021 / Published online: 3 August 2021
© The Author(s) 2021

Abstract

In the last few years, Internet of Things (IoT) systems have drastically increased their relevance in many fundamental sectors. For this reason, assuring their quality is of paramount importance, especially in safety-critical contexts. Unfortunately, few quality assurance proposals for assuring the quality of these complex systems are present in the literature. In this paper, we extended and improved our previous approach for semi-automated model-based generation of executable test scripts. Our proposal is oriented to system-level acceptance testing of IoT systems. We have implemented a prototype tool taking in input a UML model of the system under test and some additional artefacts, and producing in output a test suite that checks if the system's behaviour is compliant with such a model. We empirically evaluated our tool employing two IoT systems: a mobile health IoT system for diabetic patients and a smart park management system part of a smart city project. Both systems involve sensors or actuators, smartphones, and a remote cloud server. Results show that the test suites generated with our tool have been able to kill 91% of the overall 260 generated mutants (i.e. artificial bugged versions of the two considered systems). Moreover, the optimisation introduced in this novel version of our prototype, based on a minimisation post-processing step, allowed to reduce the time required for executing the entire test suites (about -20/25%) with no adverse effect on the bug-detection capability.

Keywords IoT testing · Model-based test Generation · Empirical study

1 Introduction

Internet of Things (IoT) systems are systems composed of web-enabled smart devices using sensors, actuators and communication hardware, to collect, send, and manage data acquired from the surrounding environment. Smart devices do most of the work without

✉ Maurizio Leotta
maurizio.leotta@unige.it

Dario Olianas
dario.olianas@dibris.unige.it

Filippo Ricca
filippo.ricca@unige.it

¹ Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS),
Università di Genova, Genova, Italy

human interaction, although humans can interact with the devices – for instance, through smartphones for inserting values and/or receiving information and alerts. Business logic (e.g. corroborated by machine learning and advanced analytics) and data storage are usually accomplished in the cloud.

IoT systems are used in a multitude of safety-critical sectors, such as aircraft, cars, medical devices, and nuclear power plants, whose failure could result in loss of life and significant damages. As a consequence, ensuring that such systems are secure, reliable, and compliant with the requirements is a fundamental task (Reggio et al., 2020).

However, assuring the quality of these systems is extremely challenging for different reasons. To name a few, *Quality Assurance Engineers* and *Software Testers* have to face: platform diversity (i.e. disparate technologies used to build them); limitations in memory, processing power, bandwidth, battery life; data velocity; lacking of consolidated quality assurance approaches specific for IoT systems; machine learning components embedded in IoT solutions that make the systems extremely difficult to test (Braiek & Khomh, 2020).

In this paper, we propose and empirically evaluate MATTER (Model-bAsed TesT gEneRator), a tool devised for semi-automated End-to-End (E2E) testing of IoT systems, where a smartphone or a Web application plays an important role to control or monitor the system execution. *E2E testing* (Leotta et al., 2016) is a type of black-box testing performed after unit, integration and system testing to verify that the whole system under test (SUT) is working in real-world conditions, i.e. connected with external systems and in normal network conditions. We focused on E2E testing since, according to many organisations (Parasoft, 2021), assembling an IoT system and testing it as a whole (e.g. through the GUI) is the simplest and effective way to ensure its quality.

Our MATTER tool follows a *model-based* approach (Utting, 2007). Produced test scripts are derived from the model of the system: such model is represented in UML with a State machine diagram and a Class diagram as, for instance, done in UniMod (Ricca et al., 2018). The goal of MATTER is to generate a test suite composed of executable test cases (i.e. test scripts) that is complete and correct with respect to the semantics of the provided model. First, our prototype tool generates a set of test paths covering at least once all the transitions of the State machine. Second, if some test paths are infeasible, it transforms them in feasible paths exploiting the notion of branch distance (McMinn, 2011). Third, it transforms test paths into test scripts. Compared to other proposals available in the literature, to the best of our knowledge, MATTER is the only tool able to generate, almost automatically, executable test cases for IoT systems.

Since, in many cases, IoT systems can be controlled or monitored from a smartphone or a Web application, our tool-based approach relies on the user interface as the primary way of interaction between the test scripts and the system under test. Clearly, when required, generated test scripts can also send input to the system's sensors and detect actions of the actuators. Note that test scripts generated by MATTER can be used 'as-are' to test IoT systems employing virtualised devices (and so to control the simulated sensors and actuators). On the contrary, to test IoT systems including real physical sensors and actuators, some additional adapters will be necessary. Produced test scripts are directly executable on the SUT because they contain testing framework commands (e.g. Selenium or Appium) able to drive a smartphone or a Web application as a real user (e.g. inserting values in fields of a form or clicking some buttons retrieved using specific locators (Leotta et al., 2021)).

The MATTER tool has been empirically evaluated with two IoT systems, using an ad-hoc validation framework based on Mutation testing (Offutt & Untch 2000), to determine the effectiveness of the generated test suites in detecting bugs. Also, the time required to execute the generated test suites has been considered in the evaluation.

The paper extends our previous preliminary work (Olianas et al., 2020) in several directions. In particular, we provide here the following extensions: (1) a more complete description of the prototype tool and the approach also thanks to several clarifying examples, (2) a novel component of MATTER able to minimise the generated test suite, and (3) a more extensive empirical study with a novel experimental object (SmartParking, a smart park management system) and novel research questions aimed at evaluating the effectiveness of the tool and the impact of the novel introduced minimisation component. To summarise, the *main contributions* of our work are as follows: (1) a semi-automated model-based approach to the generation of executable test scripts for IoT systems; (2) a prototype tool implementing our approach, freely available online; and (3) a detailed empirical evaluation of our tool employing two IoT systems showing the benefits deriving from its adoption.

The remainder of this paper is organised as follows. Section 2 describes the main elements underlying the MATTER tool and the DiaMH IoT system, used as running example to introduce our approach. Section 3 describes the proposed approach trying to abstract from the implementation details of the MATTER tool. Section 4 presents the workflow a software tester has to adopt to use our tool and the MATTER architecture. Section 5 reports on the empirical evaluation of the tool-based approach with two IoT systems, while Section 6 describes related works and Section 7 concludes the paper and sketches possible future work.

2 Elements of the approach and DiaMH running example

In this section, we first introduce the main elements of our approach and then briefly sketch DiaMH, a simplified Diabetes Mobile Health system used as a running example.

2.1 Testing IoT systems

Testing IoT systems poses important challenges, given that this kind of systems include several components working together and with a high risk of individual failure. Moreover, many faults could be due to the integration of the components or are revealed only when the system is executed as a whole. In general, a complete test plan should include a combination of unit testing (components should be tested in isolation), integration testing (components should be tested as a group), and acceptance testing using End-to-End approaches (the system should be tested as a whole by means of user actions performed on the GUI).

To be complete, the testing process should contain two phases executed in series:

1. Test a virtualised version of the IoT-system, where real smart devices are not employed, and the goal here is to test only the software¹. In this case, virtual devices have to be implemented (better if using the same software than embedded in real devices) and used for stimulating the applications under test.
2. Test the real IoT-system complete of real smart devices. The goal here is testing the system in real conditions, i.e. in real-world scenarios with hardware, network, and other applications.

¹ <https://wso2.com/whitepapers/a-reference-architecture-for-the-internet-of-things/>

Since, usually, IoT systems are safety-critical, both testing phases should be conducted, and the former could favour earlier implementation problems detection and so potentially reveal more faults. For example, because timings of sensors can be made shorter in virtual modality and thus a higher number of tests could be executed.

In this work, we focus mainly on testing phase 1 because it is the first one that a Testing team has to face, and it can be conducted without employing real sensors and actuators that could be expensive and complex to use/set. However, the test scripts generated by MATTER with virtualised devices can be later used to test the real IoT system. In this case, some adapters will be probably required to interact with real sensors and actuators. Depending on the case, such adapters can be: (1) software components in case of a software interaction with smart sensors/actuators, or (2) physical components in case of physical interaction (monitoring) with simpler physical sensors (actuators).

2.2 Elements of the approach

The main elements of our approach are: 1) formalisation of the expected behaviour of the system in a model, 2) model-based testing (i.e. test artefacts generation from the model), 3) virtualisation of physical devices and emulators, and 4) implementation of test scripts using E2E test automation frameworks.

We decided to formalise the system behaviour in terms of a UML State machine to guide the testing activities, as suggested by various model-based testing techniques, where state-based models describing systems behaviours are used for code generation and test cases derivation (Utting, 2007). And at the same time, we focus also on test automation frameworks (Dustin et al., 1999; Leotta et al., 2016) and virtual components. Indeed, test automation frameworks control the execution of test scripts, giving commands directly on the UI and reading actual outcomes to be compared with predicted outcomes. Virtual devices are pieces of software that mimic the behaviour of real devices and are fundamental for testing the system without dealing with physical objects.

2.3 DiaMH running example

DiaMH is mainly used to explain the underlying approach to the MATTER tool and is one of the two systems used in the empirical study presented in Section 5.

DiaMH is an IoT system that monitors a patient glucose level, sends alarms to the patient smartphone when a dangerous glucose trend is detected and regulates insulin dose. Two previous works have inspired DiaMH description: a Parasoft white paper (Parasoft, 2021) and the paper by Istepanian et al. (Istepanian et al., 2011).

DiaMH has already been used as a case study in other works concerning testing of IoT systems (Leotta et al., 2018) where, however, a manual approach to derive test cases from a model of the behaviour of the system has been proposed in contrast to the automated generation proposed here. As shown in Fig. 1 the DiaMH system is composed by:

- A *glucose sensor* and an *insulin pump* both worn by the patient;
- The patient *smartphone*, wirelessly connected to both the glucose sensor and the pump, and used to receive an alarm in case of dangerous blood glucose levels;
- The analysis and control system, *DiaMH Core*, running on the server side cloud, that receives the glucose readings, analyses their patterns and commands the proper actions (e.g. injecting insulin in case of high blood glucose levels, or hyperglycemia).

Fig. 1 DiaMH components, testing execution environment, and relations with test scripts

The behaviour of DiaMH is the following. The glucose sensor measures the glucose level of the patient at given timed intervals (e.g. every 30 minutes in real settings but can be reduced to few seconds when testing the system), and sends them to the smartphone. The mobile app displays the value and forwards it to the remote DiaMH Core. The core component of the system stores the last 20 glucose readings received from the patient's smartphone and, depending on how many readings exceed a given threshold (e.g. 140 mg/dL), it will command an insulin injection, and if necessary, send an alarm to the patient.

After receiving a value, the analysis and control system decides the state assigned to the user and any actions to be taken:

- *Normal*: between 0-3 values among the last 20 exceeded the threshold. No action required;
- *More insulin required*: between 4-15 values among the last 20 exceeded the threshold. An insulin injection must be ordered;
- *Problematic*: between 16-20 values among the last 20 exceeded the threshold. An insulin injection must be ordered, and an alarm is displayed on the smartphone.

To prevent performing too many injections, and to allow insulin to take effect, every time an insulin injection is performed the following five glucose readings are ignored (corresponding in our implementation at a time equal to 2.5h without injections). When an injection is ordered, the patient should confirm it using the smartphone. As a consequence, the app will command the pump to inject the insulin dose. To prevent missing injections because of pump failures, the pump has to send a feedback to the smartphone when the injection is performed successfully. After the feedback, the app will show the total number of injections performed in the current session.

A possible simple test case t for DiaMH produced by our tool (and then transformed in executable test script) could be the following: 1) t sets the glucose sensor to send between 4-15 values over threshold among the last 20, 2) t checks on the smartphone if the insulin injection has been ordered.

Figure 1 reports the architecture of DiaMH with virtualised devices (glucose sensor and insulin pump), the emulated smartphone² (running the real apps), the DiaMH cloud component, the testing environment, and the interactions among components.

3 The proposed IoT model-based testing approach

In this section, we fix the terminology and describe the model-based testing (MBT) approach we adopted to (semi) automatically generate test suites for IoT systems.

In general, a *test suite* is a finite set of test cases. A *test case* is a set of inputs, execution conditions and expected results developed for a particular objective, such as to exercise a specific program path or to verify compliance with one particular requirement (ISO/IEC/IEEE, 2010). The input part of a test case is called *test input*. A *test script* is an executable code implementing a test case that can be executed by a Testing framework (Dustin et al., 1999).

MBT is a specific software testing approach/technique in which the test cases are derived from an abstract model (or a set of abstract models) that describes the functional aspects of the system under test (SUT). Models are named *abstract* because they contain fewer details w.r.t. the SUT, and thus, they are simpler than the SUT (easier to produce/check/modify and maintain). More precisely, an MBT approach usually encompasses (Utting et al., 2012): the automatic derivation of test cases from abstract models, the generation of test scripts from test cases, and the automated execution of the resulting test scripts.

According to Utting et al. (2012), our approach to automatically generate test suites for IoT systems is composed by the following steps: test model preparation, test selection criteria definition, test cases (or test paths) generation, test scripts generation, test scripts execution. Each step is explained in detail in the following sub-sections.

3.1 Test model preparation

In MBT, it is common to develop a test-specific model of the SUT directly from the informal requirements or existing specification documents. Our approach is no exception.

Starting from existing specification documents describing the system's components and behaviour or trying to get this information in case these documents do not exist or are out of date, the software tester adopting our approach has to produce a test model for the IoT SUT. The test model is composed of a UML Class and a UML State Machine diagram plus some classes (Mock and Wrapper) and configuration files used to fill the gap between information contained in the test model and actual test script code.

In our experience, UML Class diagrams and UML State Machine diagrams are well suited to precisely describe the several different components/devices of the IoT SUT and to model features and changes of state that are relevant to express the global behaviour of the system under test.

Thus, to follow our approach, the software tester has first to create a class diagram containing a class for each component of the IoT system. The classes in the class diagram will represent in an abstract way the smart devices, i.e. the things or components,

² an emulator is a program that enables one computer system (e.g. a Windows personal computer) to behave like another system (e.g. an Android smartphone)

Fig. 2 UML Class Diagram of the DiaMH system

composing the IoT system. Then, the tester producing the class diagram has to insert in each class: fields representing the internal state of the components and operations corresponding to services offered by components. For example, if the component/device *InsulinPump* of DiaMH executes an injection to a patient, the software tester will have to insert the *InsulinPump* class in the class diagram with at least the *inject* operation. Moreover, if it is important for generating the test scripts to keep track of the number of injections the software tester will have to insert in the class at hand also a field storing it.

More precisely, each class may contain three kinds of members: (a) method operations, denoted with the visibility modifier '+', performing some internal actions and returning a value (also void), (b) operations generating call events, that will be used as events in the UML state machine (those latter operations should not return a value), and (c) fields, storing some values used by the components, and representing the internal state of the component.

Figure 2 sketches the class diagram designed for DiaMH. It contains a class for each DiaMH component, i.e. *MobileApp*, *Cloud*, *Sensor*, and *InsulinPump*. In the class diagram, there are no associations because the classes are used only to interact with the complete system, and the object instances do not need to interact with each other.

As an example, the class *InsulinPump* represents and simulates the Insulin pump device. It contains a field (*erogatedInjections*) and three operation methods (*inject*, *reset* and *getErogatedInjections*). The field *erogatedInjections* corresponds to the internal state of *InsulinPump* and keeps track of the number of erogated injections. The operation *inject* simulates an injection and increments by one *erogatedInjections*. The *reset* operation is used for testing purposes to set to zero *erogatedInjections*. Finally, *getErogatedInjections* is a getter method operation. Another class deserving an explanation is the *MobileApp* class, since this corresponds to the component on which the assertions will be checked. The method operations in the class *MobileApp* returning bool (e.g. *isAlarmed()* and *isNotAlarmed()*) will play the role of assertions in the generated test scripts.

After having produced a UML class diagram containing a class for each component of the IoT system, the software tester has to model the UML state machine describing the required behaviour of the IoT system under test. To produce the state machine, the software tester must decide the states that compose it and the transitions. Each transition has to be labelled with an *event*, a *guard* and an *action*, all possibly optional. An event can cause a state transition, while a guard is a Boolean expression that is dynamically evaluated. A guard affects the behaviour of a state machine by enabling transitions and actions only when it evaluates to true. When an event is produced and the guard (if exists) true, the state machine responds by performing some actions, such as, e.g. changing a class field or calling method operations.

More precisely, the semantics of the state machine in response to incoming events is based on the so-called run-to-completion step (Samek, 2008). When the event has been received and the guard (if present) is true the transition is enabled and the *run-to-completion step* starts: the corresponding transition is fired and no other event can be dispatched until the processing of the current transition is completed. If the transition contains actions, they are executed in sequential order, and the transition processing is not considered complete until all the actions have completed their execution.

We call test path an ordered sequence of state machine transitions, starting from an initial state, that represents a test case. An executable test path or feasible path is a path that, if traversed on the state machine, never meets a false guard and so can be executed.

Figure 3 reports the state machine designed for DiaMH: there are three states representing a generic patient's condition: Normal, MoreInsulin, and Problematic, together with all the details describing the required behaviour of DiaMH and of its components. It is important to note that the state machine is consistent with the class diagram since each event is associated with an operation generating call events of a class and each action invokes an operation method defined in the class diagram. Note that, the state machine diagram reported in Fig. 3 represents the global states of the entire DiaMH system and does not detail the states of the various components.

The execution flow of the state machine always starts from the initial pseudo-state (a black circle in the diagram), so that the transition outgoing from the initial state must contain the initialisation of all the objects used in the state machine (see Fig. 3), along with one or more methods call or variables declaration to be executed at the beginning of each generated test script.

The class diagram and state machine diagram must be carefully produced manually, and this is a fundamental aspect of the approach. The creation of these artefacts cannot be generalised for all systems; it is strictly related to the specific system under test. For example, for DiaMH, it seems reasonable to use as states the different levels of alert that can be assigned to a patient (i.e. Normal, More insulin, Problematic). Then, to define transitions, a software tester has to look at the conditions that should happen to switch from a state to another (e.g. the state machine is in the Normal state and four glucose readings over threshold arrive) and create the transition. It is important to distinguish between actions that are executed in response to an event (e.g. store the reading and increase the index, implemented by the methods Cloud.receiveUnder and Cloud.receiveOver) from actions that are executed when state's internal variables (e.g. values of attributes or returned by methods) reach some conditions. The latter actions will be represented as transitions with completion event, with a boolean expression representing the variable's state as a guard. In DiaMH's model, all transitions going from one state to another are transitions with completion event

Fig. 3 UML State Machine Diagram of the DiaMH system

(see Fig. 3): they do not happen in response to an event, but every time the cloud is idle (i.e. no reading is being received, no action is being commanded) the state machine controls its internal variables to check if the conditions expressed in guards are met.

3.2 Test selection criteria definition

Test selection criteria are chosen, to guide the automatic test generation to produce a 'good' test suite (Utting et al., 2012). In our case, they are related to the structure of the model and to data coverage heuristics.

In structural model coverage, the primary issue is to measure and maximise the coverage of the model. We opted for the transition-based coverage named all-transitions. All-transitions criterion requires that every transition of the model (in our case, a UML state machine) is traversed at least once. This coverage criterion was chosen because even if it is not the best to detect bugs, it is effective (Briand et al., 2004) and at the same time does not generate a considerable number of paths such as, e.g. the path coverage criterion.

Concerning data coverage, we opted for a heuristic/strategy able to integrate inputs directly in the model. For example, the choice of having two operations `receiveOver/receiveUnder` in DiaMH is due to the fact that the proposed approach does not support arbitrary input coming from a user or from the environment: each input value must be either fixed or computed during the execution. With *arbitrary input*, we mean an input

whose value is produced externally to the system, for example, by a user, the environment or another system. Since, usually, systems accept *arbitrary input* (and DiaMH makes no exception, values coming from sensors are an example of arbitrary input), a way to handle it must be devised. In case we have a small number of different possible inputs, we may use a solution based on equivalence partitioning. *Equivalence partitioning* (Myers et al., 2012) is a software testing technique that divides the input data of a software into partitions of values producing an equivalent behaviour. To employ equivalence partitioning in our approach, we proceed in this way. Let us suppose we have to incorporate in a class C of the model an operation *receive()* managing an arbitrary input X of type integer. First, we can create an enumeration type E having a value for each equivalence class of values of X that produce the same behaviour when given in input to the SUT. Second, for each value $(v_1, \dots, v_n)$ of E we can insert in C the operations: *receive*₁ $()$, ..., *receive* _{n} $()$ which simulate the reception of specific arbitrary values.

It is important to note that this strategy can only be used when the arbitrary input has not associated too many partitions. Otherwise, it would become impractical to add a bog number of operations to the class diagram.

3.3 Test path generation

Once the model and the test case criteria are defined, a set of test cases can be generated with the aim of satisfying the decided criteria. In practice, a test case is a test path built by walking through the UML state machine and starting from the initial state. As we built the state machine, input data and assertions are already embedded in the generated test cases.

To satisfy the decided All-transitions criterion on the state machine diagram, we first transformed the state machine into an intermediate directed multigraph, then, we applied an algorithm to compute the list of paths covering all the edges.

A *directed multigraph* G is a directed graph that is permitted to have multiple edges, i.e. edges with the same source and target nodes. In our implementation, G is represented as a set of vertices V (the states of the diagram) and a set of edges E (the transitions of the diagram), and an initial state s . We assume that every state in V is reachable from s . Each element of V contains the state itself and a path to reach it, saved the first time we visit the state. Each element of E contains the edge itself and a Boolean to mark if the edge has been visited yet.

Each vertex v and edge e offers the following methods:

- v .*hasOutEdges()*: returns **true** if the vertex has outgoing edges, **false** otherwise;
- v .*unvisitedOutEdges()*: returns the list of outgoing edges from v not yet visited;
- e .*getTarget()*: returns the destination vertex of e ;
- e .*getSource()*: returns the source vertex of e .

To compute the coverage of all edges of a graph, there exist different well-known algorithms, like the algorithms solving the Chinese Postman Problem (Thimbleby, 2003) and the k-shortest-paths algorithm (Yen, 1971). However, they are not optimal in our case. The first is because it generates the shortest closed path that visits every edge of the graph and

returns a single long path that guarantees edge coverage. Having a single test script makes the localisation of the fault in the code harder, and for this reason, we did not adopt it. The second algorithm has been implemented in (Mesbah & van Deursen, 2009) to cover the navigational graph of a Web application. However, after a deep analysis, we discovered that it generates many similar paths, thus risking to worsen the already long test execution times present in the IoT context, so we preferred to devise an ad-hoc algorithm for this purpose.

Listing 1 sketches the algorithm we devised. The rationale of this algorithm is to satisfy the coverage of all transitions avoiding to generate: (1) a single long path as in the case of the Chinese Postman algorithm (Thimbleby, 2003) and (2) many similar paths as in the case of the algorithm presented in (Yen, 1971). Our algorithm creates a new path in two cases: each time it visits an already visited node and when it visits a node without outgoing edges. Since all the unvisited edges of a node are added to the `edgeToVisit` queue (see line 10 of Listing 1) when the node is visited for the first time, if we assume that every node is reachable from the starting node *s*, every edge will appear in at least one produced path, and this guarantees the all-transitions coverage. Moreover, since the last edge is always taken from the `edgeToVisit` queue, and since there are no duplicate edges in the queue (edges are added only when a node is visited for the first time), the algorithm also guarantees that every produced path ends with a different edge.

The application of our algorithm to DiaMH produces 12 paths reported in Listing 2 (actions are inserted only to disambiguate the paths).

```

1 list FindPathsForAllTransitionsCoverage(V,E,s) {
2   Queue edgeToVisit; List finalPaths;
3   edgeToVisit.push(s.unvisitedOutEdges());
4   while (TRUE) {
5     e = edgeToVisit.pop(); vertex v = e.getTarget();
6     e.visited = TRUE;
7     if (v.path is empty) { /* the first time the node is visited, the path to
8       reach v is created by concatenating the current edge to the path
9       bringing to its source state */
10      v.path = e.getSource().path+e;
11      /* if the current state has outgoing edges they are added to the
12       unvisited queue */
13      if (v.hasOutEdges()) edgeToVisit.push(v.unvisitedOutEdges());
14      /* if there are no outgoing edges the path is finalised */
15      else finalPaths.add(v.path);
16    }
17    else finalPaths.add(e.getSource().path+e); /* This node has been already
18      visited, so we finalize the path */
19    if (edgeToVisit is empty) break;
20  }
21  return finalPaths;
22 }
```

Listing 1 Algorithm producing a list of paths able to satisfy the all-transitions coverage

States => i: initial, N: normal, MI: more insulin, P: problematic

```

P 1: i->N, N->N / cloud.receiveUnder();
P 2: i->N, N->N / cloud.receiveOver();
P 3: i->N, N->N [cloud.getNumReadings()==cloud.getMaxReadings() and 0<=cloud.
    getCriticalCount() and cloud.getCriticalCount()<4]
P 4: i->N, N->MI, MI->N
P 5: i->N, N->MI, MI->MI [cloud.getNumReadings() == cloud.getMaxReadings() and
    4<=cloud.getCriticalCount() and cloud.getCriticalCount()<=15]
P 6: i->N, N->MI, MI->MI / cloud.receiveOver();
P 7: i->N, N->MI, MI->MI / cloud.receiveUnder();
P 8: i->N, N->MI, MI->P, P->MI
P 9: i->N, N->MI, MI->P, P->P [ cloud.getNumReadings() == cloud.getMaxReadings
    () and cloud.getCriticalCount() > 15]
P 10: i->N, N->MI, MI->P, P->P / cloud.receiveOver();
P 11: i->N, N->MI, MI->P, P->P / cloud.receiveUnder();
P 12: i->N, N->MI, MI->P, P->N

```

Listing 2 12 paths generated by our algorithm for DiaMH. Actions are inserted only to disambiguate the paths.

3.3.1 Test scripts generation

Test paths generated in the previous step are built looking only at the graph structure of the UML state machine, without checking if transitions' guards are enabled, and so the paths are feasible during execution. Therefore, they cannot be transformed immediately in executable test scripts. We need a further step to check whether the generated paths are executable.

To execute a test path, we have to start from the first transition and execute all the code included in transition labels (i.e. events, guards, and actions) and entry actions (if any) of the target state. Since the code in the state machine refers to classes declared in the class diagram, to be able to execute test paths, we have to add to our approach another element: mock classes (not to be confused with virtualised devices implemented with Node-RED, see Section 5.2). The only purpose of mock classes is to simulate the behaviour of IoT components to be used during test case generation. Thus, mock classes have to be manually designed by the software tester to make sure their behaviour is compliant with the components of the real IoT SUT. However, model-based code generation techniques can be employed to partially generate them from the UML class diagram.

As an example, we report in Listing 3, a portion of the Python code of the Cloud mock class we designed to generate the test suite for DiaMH.

```

1 class Cloud:
2     def getCriticalCount(self):
3         res = 0
4         for x in self.glucoseReadings:
5             if (x > self.threshold):
6                 res += 1
7         return res
8 ...

```

Listing 3 Portion of the Python code of the Cloud mock class

The operation `getCriticalCount(...)` returns the number of glucose readings exceeding the threshold received by the cloud so far. It is invoked every time the test generation tool traverses a transition which, in its guard, has to check the number of values over the threshold.

To transform produced test paths in executable test scripts, we have to render them feasible, i.e. test paths that, during execution, have all the guards true. The executability problem is in general undecidable (Bourhfir et al., 1997). However, in most cases, it can be solved with some heuristics. To choose which transitions traverse to make the guards true, we used the notion of *branch distance*. Described for the first time by Korel (1990), the branch distance is a metric that can be computed on Boolean predicates, and tells us how close we are to make the given predicate true. The original version of Korel handled only relational predicates (i.e. $=$, $<$, $\leq$, $>$, $\geq$), but it has been extended by Tracey et al. (1998) to support negation, AND and OR. Usually, the branch distance is used as a fitness function (or as a component of the fitness function) in meta-heuristic search algorithms like hill-climbing, simulated annealing or genetic algorithms for the automatic generation of test data (McMinn, 2011). The fitness function is used to guide the search to good solutions from a potentially infinite search space, within a practical time limit.

In our specific case, the goal is simpler; our algorithm (named executability checker) has to find a *loop* (i.e. a path returning on the source node) in the state machine (if any), with a fixed maximum length³, able to enable the transition having the guard false. So at each step, the search space is limited by the number of outgoing transitions of the current state; so our search algorithm is quite straightforward.

Let us make an example on the DiaMH state machine (see Fig. 3) to clarify the previous statement. Let us suppose, we have to enable the transition from the **Normal** to the **More-Insulin** state and we have already received 10 readings of which two above the threshold (i.e. critical readings). In this case, we have to enable the transition guard `[cloud.getCriticalCount()>=4]` that is false. Tracey et al. (1998) report that the branch distance for the predicate $E1 \geq E2$ is 0 if $E1 \geq E2$ otherwise $E2 - E1$. Thus, the current distance d is $4 - 2 = 2$. Our algorithm has to analyse the effect of the guard of interest on the branch distance executing all the transitions outgoing from the **Normal** state: `cloud.receiveOver()`; increments of 1 the value of `criticalCount`, `cloud.receiveUnder()`; does not modify the value of `criticalCount`, while the remaining self-transition of the state **Normal** cannot be executed since `maxReadings` is equal to 20 and we have obtained only 10 readings so far. So to reduce the distance d to our goal (i.e. enabling the transition from **Normal** to **More-Insulin**), the algorithm has to choose to cycle twice on the self-transition labelled with `cloud.receiveOver()`. Indeed, in this way, d changes from 2 to 1 (after the first cycle), and then to 0 (after the second cycle). At this point, `criticalCount` becomes equal to 4 and the transition to **MoreInsulin** can be executed.

If we apply the executability checker to the test path 4 of DiaMH (i.e. `init` $\rightarrow$ **Normal**, **Normal** $\rightarrow$ **More insulin**, **More insulin** $\rightarrow$ **Normal**), that is clearly infeasible since it meets two false guards (e.g. to execute the second transition, `cloud.getCriticalCount()` must be greater or equal than 4) we obtain the feasible test path show in Listing 4.

Clearly, the executability checker implements a heuristic that, in some cases, can fail. When it fails, the infeasible paths produced during test case generation are discarded, and the transition coverage of the state machine will not be total. However, in practical cases,

³ this maximum value depends on the characteristics of each specific IoT system under test and of the state machine model and should be selected by the software tester.

```

States => i: initial, N: normal, MI: more insulin, P: problematic
i --> N
N --> N / cloud.receiveOver() (executed four times)
N --> MI
MI --> MI / cloud.receiveOver() (executed eight times)
MI --> MI / cloud.receiveUnder() (executed 17 times)
MI --> N

```

Listing 4 Example of feasible path for DiaMH

we have seen that carefully designing the state machine is possible to achieve high coverage levels.

3.3.2 Test suite minimisation

A side effect of the proposed approach is that the generation algorithms may generate many redundant test paths. We call redundant test path a test path that is included in another one: for example, the path $A \rightarrow A, A \rightarrow B$ is included in the path $A \rightarrow A, A \rightarrow B, B \rightarrow C$. Generating and then adding redundant test cases to a test suite does not improve its fault detection capability since each command executed by included test cases is already executed from the test case, which includes all the redundant test cases. Moreover, generating redundant test cases makes the test suite bigger and increases its execution time. Therefore, we introduced a minimisation step in the **MATTER** tool that eliminates redundant test paths before generating test scripts: it consists of a sorting step, where test paths generated after executability handling are sorted by length, and a minimisation step where, for each test path, we check if it is contained in subsequent test paths, and if this happens the path is removed from the result.

3.4 Test scripts execution

At this point, almost all the information required to have fully executable test scripts is ready. We have only to fill the gap between test paths generated in the previous step and test code executable in a specific programming language for a specific Testing Framework.

We filled this gap resorting to another element, named wrapper classes, and to pre-compiled configuration files. A test script has to perform a sequence of interactions with system components, retrieve from the components some information about the system's status and check if the retrieved information is correct by means of an assertion. In a realistic scenario, none of these steps can be done with a single instruction of a high-level programming language like Java or Python. Interactions with the IoT SUT usually happen over different communication protocols (MQTT, TCP, Bluetooth), and typically require additional libraries (such as Paho for MQTT, Appium to control Android devices and JUnit to check assertions that can be written using specific libraries (Leotta et al., 2020)).

To avoid writing long pieces of code onto the model and simplify model's automatic analysis, it is useful to encapsulate in wrapper classes the code that performs the operations on the components required for testing. Wrapper classes have to be implemented by the software tester following the class diagram defined in the modelling phase.

```

1  @test
2  public void test_4() {
3      // transition    0: init    -> Normal
4      ... initialisation
5      // transition    1: Normal  -> Normal
6      sensor.sendOver(cloud.getThreshold());
7      cloud.receiveOver();
8      // transition    2: Normal  -> Normal
9      ...
10     // transition    3: Normal  -> Normal
11     ...
12     // transition    4: Normal  -> Normal
13     ...
14     // transition    5: Normal  -> More insulin
15     app.assertRequiringInsulin(timeout);
16     app.assertAlarmOff(timeout);
17     app.confirmInsulin();
18     // transition    6: More insulin -> More insulin
19     ...
20 }

```

Listing 5 Example of the initial portion of a test script for DiaMH derived from the test path reported Listing 4

So, to transform the test path shown in Listing 4 in the test script shown in Listing 5, we just have to traverse the path and copy in sequential order the code found in transitions and states. This code will refer to the aforementioned wrapper classes to interact with the system under test.

For example, the App wrapper class will contain the `assertAlarmOff(int timeout)` operation, corresponding to the model's method `isNotAlarmed`, called at transition 5 of test script 4. This operation will use the Appium API to communicate with the DiaMH Mobile App interface and the JUnit assertion to check that the alarm button is not displayed on the GUI.

This decoupling between test logic (contained in the test paths) and implementation details (contained in the wrapper classes), as well as being useful in the test generation phase, it also allows running test scripts against different IoT SUT implementations simply by changing the wrapper classes. To give more concrete examples, in DiaMH we may have different versions of the system: one where the communication protocol is MQTT, one where is TCP, one where the devices are virtualised, one where the devices are real. All these variants can be tested with the same test scripts, adapting the wrapper classes.

4 MATTER tool

The workflow a software tester has to adopt to use MATTER tool, along with the involved artefacts and the relations among them, is shown in Fig. 4.

To summarise, starting from the informal requirements or existing specification documents, the software tester has to design a testing model for the system, namely a UML

Fig. 4 Approach workflow

Class Diagram that describes its components and a UML State Machine diagram that illustrates the behaviour of the core components.

Then, the model is used to derive two sets of classes to be implemented: wrapper classes and mock classes. Wrapper classes are used by test scripts to interact with components, mock classes are used by the test generation tool to generate executable test paths. Wrapper and mock classes must be manually implemented: the first ones in the language used to implement test scripts, the latter in Python, since they have to be executed by our tool, which is written in Python.

Interfaces of the wrapper and mock classes, although derived from the model, can significantly differ from the interfaces of the IoT system components. To fill the gap between model and implementation, some *configuration files* must be *defined*. In particular, we decided to adopt four configuration files:

1. *Implementation definition file*: associates every class in the model with the actual implementation we want to use in test scripts (e.g. for DiaMH it contains only a JSON object where each class in the model is associated to a concrete wrapper class implementation { "MobileApp" : "MobileAppAppium", "Cloud" : "Cloud", "Sensor" : "GlucoseSensor", "Pump" : "InsulinPump" }). At this prototyping stage, it must be defined manually, but in a real-world scenario, it may be generated by a tool that lets the developers choose which implementations they want to use in test scripts;
2. *Wrapper class configuration file*: maps the interface of classes in the model with the interface of the implementations. It can be automatically generated from wrapper classes;
3. *Variable definition file*: contains the information required to instantiate the variables of the state machine in the generated test scripts (e.g. for the app variable: "app" : { "type" : "MobileApp", "options" : { "modifier" : "static", "visibility" : "protected" } }). At this prototyping stage it must be defined manually, but in a real-world scenario the information about type and access modifier of variables may be integrated in the model;
4. *Language configuration file*: maps some constructs of the pseudocode of the state machine in the actual code of test scripts (e.g. in the state machine we declare objects as `VarName := Class()`, while in Java the syntax is different `Type VarName = new Class();`). Since this file depends only on the target language and the pseudocode, it is provided with the test generation tool for the supported target languages (Java and Python).

Fig. 5 Tool architecture and execution workflow

We chose to use four files instead of saving all this information in one to allow changing them independently when test scripts have to be generated. For example, if we want to use a different SUT implementation (e.g. DiaMH over TCP instead of DiaMH over MQTT), we need to change only the implementation definition file.

When all the elements shown in Fig. 4 are ready, the software tester is just required to run the tool, providing them as input. The tool will parse the state machine description and will build a set of paths that traverse it, starting from the initial state. These paths, using all the elements previously described, are transformed by the tool in executable test scripts in a given target language (for now, only Java and Python languages are supported).

4.1 Tool architecture and execution workflow

The architecture of MATTER and its execution workflow are described in Fig. 5. The tool reads the state machine, parses its code and finds an initial set of test paths using our algorithm that guarantees transition coverage (every transition of the state machine is traversed at least once considering all the generated paths).

The configuration builder (Step ①) inspects the wrapper classes (created starting from the Class Diagram) using reflection and creates the Wrapper class configuration file. Then, the Parser (Step ②) reads the state machine description and using the Wrapper classes configuration file, and the Implementation definition file creates the intermediate directed multigraph used in the subsequent step. Our path generation algorithm (Step ③) takes in input it and generates a set of paths traversing the graph, satisfying the all-transition criterion. These paths first, using Mock classes, are transformed into executable test paths or feasible paths (when possible) by the executability checker (Step ④), then are minimised by the minimiser module (Step ⑤), and finally, the output of the previous step is converted

in executable test scripts by the Test case builder (Step ⑥) in the chosen target language (our implementation currently supports Java and Python) and it relies on Appium as the framework used to automate the execution and to interface with the mobile application.

For the interested reader, further details about the MATTER tool, its source code, and the complete model and artefacts of the DiaMH case study are available at: <https://sepl.dibris.unige.it/MATTER.php>.

5 Empirical study

This section describes the design, experimental objects, research questions, metrics, validation framework, procedure, results and threats to validity of the empirical study conducted to evaluate MATTER. We follow the guidelines by Wohlin et al. (2012) on designing and reporting empirical studies in software engineering.

5.1 Study design

The *goal* of the empirical study is to *measure the effectiveness of the test suites generated by MATTER and the effect of the minimisation process* with a particular focus on: (1) the capability of the generated test suites to detect bugs in the IoT systems under test, and (2) the time savings due to test suites minimisation.

The results of this study can be interpreted from multiple *perspectives*: Researchers, interested in empirical data about the effectiveness of a model-driven tool aimed at generating test scripts for IoT systems; Software Testers and Project/Quality Assurance Managers, interested in evidence data about the benefits of adopting MATTER in their companies. The *experimental objects*, used to experiment MATTER are the two IoT systems described in the next section.

5.2 Experimental objects

To validate the proposed test generation approach, two IoT systems have been used: DiaMH (Istepanian et al., 2011; Leotta et al., 2018) and SmartParking (SmartSantander, 2012; Sanchez, 2014). Their core components are both developed in Node-RED.

Node-RED (<https://nodered.org/>) is a visual flow-based programming tool built on the Javascript server-side runtime environment Node.js, which allows developing applications as a flow of interconnected nodes. The developer uses a web-based flow editor to place different types of nodes in the flow and deploy it. The execution model is based on *nodes* and *messages*: a node is a functional unit of the flow delegated to a specific task. A message is a JSON object exchanged between nodes, that will perform some actions over its properties. It can be received from the network or generated by a node.

The major characteristics of the models of the two systems are reported in Table 1. Although both are small systems, from the table it is evident that SmartParking assumes a larger number of possible states during its execution. In the following, we provide a brief description of SmartParking.

Table 1 DiaMH and SmartParking UML Models details (SM = State Machine and CD = Class Diagram)

System	SM States	SM Transitions	CD Classes
DiaMH	3	15	4
SmartParking	9	52	4

5.2.1 DiaMH

DiaMH is the Diabetes Mobile Health system that has been already introduced in Section 2. Figure 1 describes the architecture of the system with virtualised devices (glucose sensor and insulin pump), the emulated smartphone (running the real apps), the DiaMH cloud component, the testing environment, and the interactions among components. The sensor and the pump components are virtualised by Node-RED flows, the cloud component has been natively developed in Node-RED, while the mobile app is running on an Android emulator (note that the code of the mobile app is the same used in the real DiaMH system).

5.2.2 SmartParking

SmartParking is a smart park management system inspired by the one presented in the SmartSantander project (SmartSantander, 2012; Sanchez, 2014). Users can employ a mobile app installed on their phone to ask the system if there are free parking lots and, in the affirmative case, see them reported with the nearest first: individual parking lots are monitored by sensors that tell to the central server if each place is taken or not. There are three types of parking lots: for cars, for motorcycle and for disabled people.

We developed a slightly simplified version of this IoT system, where all the components, except for the mobile app, are emulated in Node-RED. The test execution environment is very similar to the one used for DiaMH: test scripts send messages to the sensor manager running in Node-RED, and interact with the mobile application executed on the Android Emulator to see if the correct state of parking lots is displayed.

In the SmartParking implementation, a class **ParkingSpace** represents a single parking lot, that will have its position (expressed as **GPSCoordinate**), type, current status (free or taken) and a unique ID. Number of total parking lots per type and occupied parking lots per type are stored in **SmartParkingServer**, respectively in **max < type >** (number of existing parking lots for **< type >**) and **n < type >** (number of currently taken parking lots for **< type >**). There are **maxCar+maxMoto+maxHandicap** instances of **ParkingSpace**, and they all communicate with **SensorManager** whenever their status change, calling the proper service (e.g. **newCar**, **carLeaves**). The **SensorManager** will notify the change to the **SmartParkingServer**, that will update its variables consequently. Finally, the **Mobile-App** can ask the **SmartParkingServer** if there are parking lots of a certain type available in the surroundings of a given position.

A critical decision to be made to build the state machine, which then impacts the number and quality of test scripts generated, is that of the system states. Indeed, if the state machine is represented in too much detail, we may end up with a huge number of states (*state explosion* problem), that heavily slows down test scripts generation and execution. Therefore, the software tester has to choose the right level of detail carefully. To address the state explosion problem, we have decided to adopt an approach that we could define as

a 'parking type' granularity level. Since there are three types of parking, if we aggregate by type, we will have overall only 8 ($2^3 = 8$) states in the state machine.

- All parking lots are free (all free)
- Car over, free moto, free handicap
- Free car, moto over, free handicap
- Free car, free moto, handicap over
- Car over, moto over, free handicap
- Free car, moto over, handicap over
- Car over, free moto, handicap over
- All parking lots are taken (all over)

The complete model of SmartParking system used for the generation of the test scripts, including the UML StateMachine and the Class Diagram is available in the online replication package: <https://sepl.dibris.unige.it/MATTER.php>.

5.3 Research question and metrics

Our study aims at answering the following research questions:

- RQ1:** What is the effectiveness of the generated test suites in detecting bugs/faults in the IoT systems under test?
- RQ2:** Is the execution time reduction, due to the test suite minimisation step added in MATTER, significant?

To answer our research question *RQ1*, we have to quantify the bug-detection capability of the test suites generated by MATTER. Since measuring bug detection capability makes no sense for correct systems, as in the case of our two implementations used in the experiment, we decided to simulate bugs using the technique known as *Mutation Testing* (Offutt & Untch, 2000) (a possible solution to validate the effectiveness of a test suite). Mutation testing is a technique that consists in exercising the test suite against slight variations of the original code, simulating the errors a developer could introduce during development and maintenance activities. These variations of the original software system, named *mutants*, are used to identify the weaknesses in the test artefacts by determining the parts of software that are poorly or never tested. For each mutant, the test scripts are run: if at least one test script fails, the mutant has been detected (killed), and this proves the effectiveness of the test suite in detecting the kind of fault introduced by the mutant. If no test fails, the mutant is not detected (i.e. it survives), and this proves the test suite weakness in detecting the kind of fault introduced by the mutant. Thus, we decided to measure the bug-detection capability of MATTER as the number of bugs detected by each test suite over the total number of bugs present in the generated mutants. In particular, the metric we used to evaluate the overall test suite quality is the percentage of mutants killed out of the total (i.e. the higher, the better).

Instead, to answer our research question *RQ2* we have decided to measure the time (expressed in seconds) required to execute the generated test suites (both complete and minimised) on each IoT system. In particular, the metric we used to evaluate the contribution of the minimisation process is the percentage reduction from the time required to execute the complete test suites, w.r.t. the minimised ones.

Fig. 6 Execution workflow of the validation framework. Generation of the IoT systems mutants (concerning both Javascript and SQL) and mutant-based evaluation of the test suites generated by MATTER. Note that each mutated version of the IoT systems contains only a mutant per evaluation

5.4 Validation framework

To the best of our knowledge, no automated test suite validation framework exists for IoT systems based on mutation testing. Therefore, we decided to implement a support tool capable of generating the mutants of DiaMH and SmartParking, and then, for each mutant, run the test suite generated by MATTER and automatically collect the results.

Our mutation strategy is based on the extraction and mutation of JavaScript functions and SQL query contained in the Node-RED components of the two IoT systems used in the experiment. The execution workflow of the implemented validation framework is presented in Fig. 6 (upper and middle part). The starting point is the Node-RED sources of the IoT system under test. The tool ①J extracts code from function nodes of the system flows (that are the actual implementations of the system components virtualised or natively implemented with Node-RED) and saves them to file, then ②J generates mutants of the extracted code, and ③ for each mutant M of a node N , generates a copy of the Node-RED flows with M in place of N . The Javascript Mutator sub-component relies on *Stryker* (<https://stryker-mutator.io/>), a mutation tool for Javascript able to support numerous mutation operators for unary, binary, logical and updates instructions, boolean substitutions, conditional removals, arrays declarations, and block statements removals. Then, the Mutant manager and Test runner component ④ communicates with the Node-RED server and for each generated mutated Node-RED flow: (a) starts the Node-RED server executing the mutated flow; (b) runs the generated test suite against the mutated IoT system; (c) saves results; and (d) stops the server.

Moreover, to mutate the SQL select statements used by the SmartParking Node-RED server to retrieve information about parking lots from the database, we relied on SQL-Mutation (Tuya et al., 2006; Tuya et al., 2007), a tool that allows to automatically generate mutants of SQL queries using different mutation operators such as clause mutations,

operator replacements, NULL handling mutations and identifier replacements. Looking at Fig. 6 (bottom), after extracting the SQL queries used by function nodes (1S), we used SQLMutation to generate the mutated queries (2S). The mutated queries are inserted back in copies of the Node-RED flows of SmartParking server by an automated script (3), and are subsequently used by the Mutant manager and test runner tool (4) as seen for the JavaScript mutants described before. Note that each mutated version of the IoT systems contains only a mutant per evaluation.

5.5 Procedure

As the first step to answer the RQs of our study, we generated with MATTER two test suites (complete and minimised) for each of the considered IoT systems (DiaMH and SmartParking).

Then, for answering RQ1, we employed the aforementioned validation framework for each test suite and IoT system. To be as conservative as possible, we marked a mutant as killed only if it has been killed by multiple test scripts of the generated test suite. Additionally, in case only a test script of the test suite kills the mutant, then we manually re-executed the test suite against such mutant three additional times to be sure that such single killing is not due to flakiness (i.e. so the mutant must be killed in all the four executions of the test suite). This has been done to avoid marking as killed mutants that have had problems during their execution due to the complexity of the system infrastructure of the IoT systems that could potentially introduce flakiness (Eck et al., 2019)⁴. Indeed, during the mutation testing of DiaMH and SmartParking, on the system used to run the experiments (a laptop featuring an Intel Core i7 7500U, dual-core 2.90 GHz, 16 GB RAM, and Windows 10) have been running concurrently: (1) the mutant runner on Eclipse IDE, (2) the Node-RED server on a VirtualBox virtual machine (using Debian 9) with 8 GB of dedicated RAM, and (3) the mobile application on Android Emulator. The various IoT systems' components are connected through LAN, and communicate through various protocols (e.g. the MQTT protocol is used in DiaMH to transmit glucose readings and dispatch commands). Then, we manually analysed the surviving mutants to understand which are equivalent to the original version of the IoT system (i.e. mutants behaviourally equivalent to the original program). At this point, to answer RQ1, we counted how many mutants have been killed by the generated test suite considering only the non-equivalent ones, for each test suite and for each IoT systems.

To answer RQ2, we executed both the complete and the minimised test suites against the original version of the two IoT systems, selected as experimental objects, noting down the time required to complete the execution. Note that the complete test suites are generated by MATTER by deactivating the minimisation module. To minimise any fluctuation due to possible active processes during the calculation, we averaged the obtained value three times. We have estimated that this number is sufficient since we noticed that the variance is minimal.

⁴ Flaky tests exhibit a seemingly random outcome (pass or fail) when run against the same, identical software

Table 2 DiaMH mutation testing: mutant detection of the two generated test suite (complete and minimised)

Component	Mutants			Survived	Killed
	Total	Equivalent	Different		
Sensor	20	7	13	2 (15%)	11 (85%)
Cloud	116	35	81	3 (4%)	78 (96%)
Pump	49	18	31	11 (35%)	20 (65%)
Total	185	60	125	16 (13%)	109 (87%)

5.6 Results

RQ1: Effectiveness in Detecting Bugs/Faults

DiaMH For this IoT system, MATTER generated a test suite composed of 12 test cases overall that are reduced to 8 after the minimisation. To evaluate them, we employed the validation framework that provided the results (mutants killed or survived by the two test suites) for each of the 185 mutants generated for DiaMH. Since, as expected by the minimisation process, results concerning the bug detection capability of the two kinds of test suites are identical, we will describe them together. We recall that the only purpose of minimisation is that of eliminating redundant test paths before generating test scripts.

First, we analyzed in detail why some mutants have not been killed. Of the 76 surviving mutants, 60 are undetectable (see column Equivalent in Table 2) since they manifest exactly the same expected behaviour of the original DiaMH (i.e. they are equivalent mutants), and therefore they cannot be detected with any black-box testing technique (Grün et al., 2009; Jia & Harman, 2011). For example, the majority of the equivalent mutants (43 mutants out of 60) affected only log print statements on the Node-RED server console. Since logs are not described in the expected behaviour of DiaMH, mutants affecting such statements are equivalent to the original system from a functional point of view. Other mutants are equivalent because of redundant statements (12 mutants of 60): some commands performed in a function node part of the flow implementing the component (usually loading a global variable and assigning its value to an attribute of the message) are performed identically in a subsequent node before any use; so modifying these commands does not affect execution results. Finally, the remaining five equivalent mutants modify the expressions that generate glucose readings. In the sensor Node-RED component simulating the real glucose sensor, there are two operations that are used to generate values, respectively, under or over the threshold. These mutants modify some operators contained in these two operations, but the result continues to be in the right range.

Table 2 shows the number of killed/survived mutants after the execution of the two generated test suites against the remaining 125 mutants (i.e. the not equivalent ones, 185 total mutants - 60 equivalent).

Among them, 13 mutants affect the sensor component, 81 the cloud component, and 31 the pump component. The first column indicates which *Component* has been mutated, the column *Mutants* indicates the number of generated mutants for that component, while the *Survived* column indicates how many mutants were not detected by the generated test suite, the *Killed* how many were detected. From Table 2, it is evident that the generated test suites have been able to kill/detect the 87% of the mutants overall

Table 3 SmartParking mutation testing: mutant detection of the two generated test suite (complete and minimised)

Component	Mutants			Survived	Killed
	Total	Equivalent	Different		
Sensor	10	0	10	2 (20%)	8 (80%)
Server	57	1	56	0 (0%)	56 (100%)
SQL	84	15	69	5 (7%)	64 (93%)
Total	151	16	135	7 (5%)	128 (95%)

(109 out of 125). For mutants localised in the Cloud component (78 out of 81), the detection rate is even higher, reaching the 96%.

Now let us analyse why some mutants survived. In total, only 16 mutants survived. Fourteen of them change the behaviour of DiaMH but only on minor aspects that were not documented in the requirements and thus in the model. Therefore, they modify the behaviour of the original system in a way that the test scripts cannot detect the change. For example, in the insulin pump, some mutants prevent the pump from replying to pings coming from the smartphone. If the mobile app, when started, does not receive a response to the ping, a red FAIL message will appear in the pump status info on the mobile app. Nevertheless, our test scripts do not check this status info since, in the designed state machine, no assertion method checks them, and so the generated test scripts cannot detect these mutants. The test suite was unable to identify the last two survived mutants since they require a specific input to be detected. Indeed, they changed a '>' in '≥' in two Node-RED nodes checking how many readings over the threshold are stored in the cloud's memory. To appreciate a difference with the original system, we should have sent input sequences that contain exactly the threshold value.

Now let us analyse why the distribution of mutants that survived among components is unbalanced towards the Pump component (i.e. 35% vs. 15% and 4%). The reason is again to be found in what was said before: a high percentage of 'undetectable' mutants have been produced precisely for the Pump component since test scripts do not check the aforementioned status info, since in the designed state machine, no assertion method checks them.

SmartParking For this IoT system, MATTER generated a test suite composed of 43 test scripts overall that have been reduced to 31 after the minimisation step. As done for DiaMH, to evaluate them, we employed the validation framework that provided the results (killed or survived for each test scripts contained in the test suites) for each of the 151 mutants generated for SmartParking. Since, as expected, also for this IoT system the results concerning the bug detection capability of the two test suites are identical, we will describe them together.

As done for DiaMH, also for SmartParking we first analysed why some mutants have not been killed. Of the 23 surviving mutants, 16 are undetectable (see column Equivalent in Table 3) since they manifest exactly the same expected behaviour of the original SmartParking (i.e. they are equivalent mutants). More in detail, 15 of them are SQL equivalent mutants, that means queries that, in our application's domain, will always return the same data of the original query. Note that the SQLMutation tool we used offers a feature that automatically removes equivalent mutants. However, it can remove only mutants that are always equivalent, i.e. queries that will always return the same data of the original query in

every possible state of the database. The equivalent queries that we found in our analysis are instead equivalent because of the requirements of the Parking application domain. As an example, the query retrieving a user from the database for authentication is `SELECT * FROM users WHERE username = "user" AND password = "psw"`. A mutant generated by SQLMutation for this query, for example, adds a `DISTINCT` clause to it. The two queries are not universally equivalent, but they will behave differently only if in the users' table there are two users with the same username and password, and this cannot happen in our application domain. The remaining equivalent mutant (1 of 16) changed a line in the authentication node (part of the flow implementing server component) in a way that makes no difference w.r.t. the original one.

Table 3 shows the number of killed/survived mutants after the execution of each of the two generated test suites on the remaining 135 mutants (i.e. the not equivalent ones, 151 total mutants - 16 equivalent). Among them, 13 mutants affect the sensor component, 56 the server component, and 69 SQL statements.

From Table 3, it is evident that the generated test suites have been able to kill/detect the 95% of the mutants overall (128 out of 135). For mutants localised in the server component (56 out of 135), the detection rate is absolute (100%). In total, only seven mutants survived. Such non-equivalent survived mutants can be motivated as follows: the two survivors in the sensor component changed a parameter that prevents the output of the query from being returned. However, the query is executed anyway, and since it is an `UPDATE` query, its result is not interesting for the application that will behave in the same way as the original. The five survivors in the SQL mutants are queries that return different data w.r.t. the original one, but the application does not vary its observed behaviour. An example is the login query: there is a mutant that removes the username check, returning all the users with a given password (`SELECT * FROM smartparking.users WHERE psw = "password"`). This query will return all the users that have the same password, and therefore is not equivalent. But the authentication node (part of the flow implementing the Server component) of SmartParking does not check if more users are returned, it just limits to check if the query result is not empty: if it is empty, the login will fail; otherwise, it will pass. Therefore, even if this mutant is not equivalent, our test suites are not able to detect it.

Summary the generated test suites were able to detect 237 of 260 mutants; overall (91%) considering both IoT systems. Among the 23 mutants that were not detected (9%), some survived since they changed the behaviour of the systems in minor and non-documented aspects. So, to all intents and purposes, these can be undetectable mutants since the test model used for generating the test scripts did not incorporate such details. Excluding also these 14 mutants – since this is a limit more of the test model than of the approach – the mutants detection rate of our approach arrives to the 96% (i.e. 237 out of 246 mutants). It is important to note that due to the way the minimisation component was developed, the bug detection capability of the test suites is not affected by the minimisation process.

RQ2: Execution time reduction from the minimisation process

Table 4 shows the time required to execute each test suite generated by **MATTER** for both the considered IoT systems (column Time, measured in seconds). The Table also reports the difference between the complete version of the test suite and the minimised one, both as absolute value (column Difference) and percentage of reduction (column Reduction).

Table 4 Time required for the execution of the four test suites generated by MATTER

Test Suite	Test Cases	Time	Difference	Reduction
DiaMH Complete	12	975s		
DiaMH Minimised	8	785s	190s	-19.5 %
SmartParking Complete	43	8663s		
SmartParking Minimised	31	6576s	2087s	-24.1 %

From the table, it is evident that minimising the test suites, that as seen before, consists of reducing the number of test cases by removing the redundant ones (see the number of test scripts for each test suite in column Test Cases), has a positive effect on the overall time required to execute the test suite. In the case of DiaMH the reduction is of 190s from a total time required of 975s, which is equivalent to a -19.5% time reduction. Even better results are obtained in the case of SmartParking with a reduction that reaches the -24.1%; given the long execution times of the test suites generated for this app, this corresponds to a reduction of the time required of about 35 minutes (i.e. 2087s).

The execution of the SmartParking test suite requires a significant amount of time with respect to DiaMH not only because its test suite is composed of more test scripts, but also because the test scripts are constituted by more steps, and so they are more complex. This is due to two reasons:

1. The state machine of SmartParking is more complex than DiaMH's one, and therefore a higher number of test cases is required to cover it together with the fact that statistically some generated test scripts were found to be longer;
2. There are more interactions with the mobile application w.r.t. DiaMH, and therefore test scripts will last longer.

The second reason is due to the fact that in DiaMH the mobile application is by far simpler than the SmartParking's one: in SmartParking we have three different maps that show the locations of the three types of parking lots. Interacting with the application to check if parking lots are in the correct state requires to use several waiting times to prevent flakiness. Since flakiness poses a serious threat to the mutation testing validation, we followed a conservative approach inserting waiting times before every interaction in the mobile app, and this contributed to increase the duration of test scripts. However, since wait times affect both the complete and the minimised test suite of the same proportional amount, this is not a problem for the comparison.

It is important to note that, as described in Section 3.3.2, the optimisations do not influence the effectiveness of the generated test suites since only redundant test scripts are removed. We observed this effect also in practice, and indeed, the obtained results concerning the effectiveness of the minimised test suites confirm the theoretical intuition. The minimised test suites show exactly the same effectiveness in detecting bugs, so their scores are exactly the same as the complete test suite reported in the results for RQ1.

Summary the execution of the generated test suites required tens of minutes, for DiaMH, and a few hours for SmartParking. The effect of the minimisation, introduced in the novel version of MATTER, on the time required to execute the test suites is significant for both IoT systems since the reduction range from 1/5 to 1/4 of the total time (with no negative effect on the bug-detection capability).

5.7 Threats to validity

The main threats to validity affecting an empirical study are: Internal, External, Construct, and Conclusion validity (Wohlin et al., 2012).

Internal Validity threats concern confounding factors that may affect a dependent variable (in our case, number of detected mutants for RQ1 and time required for RQ2). In this context, the main threat is probably related to the choice of the tools for executing the mutations. Indeed, different tools could be potentially able to generate different mutants and so impacting the behaviour of the two IoT systems differently. This could, potentially, reduce the mutant-detection capability of the test suites generated by MATTER. To reduce this threat, we selected two mature mutation tools already used in other scientific works, one for JavaScript and the other for SQL, able to generate various kind of mutants (e.g. Stryker supports more than 30 types of mutations). Stryker has been used in (Chekam et al., 2020; Escobar-Velázquez et al., 2020) while SQLMutation in (Kaminski et al., 2011; Tuya et al., 2008).

External Validity threats are related to the generalisation of results. Both the IoT systems employed in the empirical evaluation of MATTER are examples of real systems even if small in size. Indeed, DiaMH was already available from previous works (Istepanian et al., 2011; Parasoft, 2021), while SmartParking has been created by following the specifications provided in the SmartSantander project (SmartSantander, 2012; Sanchez, 2014). This makes the context quite realistic, even though further studies with existing, more complex systems are necessary to confirm or confute the obtained results. In particular, we believe that experiments with more complex and real industrial IoT systems are needed to understand how MATTER can manage larger test models, particularly concerning the number of test scripts generated and thus the feasibility of producing and executing larger test suites. Finally, we considered two IoT systems characterised by a precise and repeatable behaviour (in other words, they are deterministic): this allowed MATTER to generated very precise assertions and thus to reach a very high mutants detection rate. In practice, IoT systems often include complex computations performed on the server-side cloud, and based on machine/deep learning algorithms. In these cases, the definition of the behaviour of the system and the testing phase (Braiek & Khomh, 2020) becomes more complex since the business logic running on the server cloud could potentially provide different results also re-executing the system with exactly the same input. As an example, let us imagine a machine learning component that continues to refine its predictions based on the input received from all the patients of the system (online machine learning (Shalev-Shwartz et al., 2011)); in this specific case, even the execution of a test script could modify the behaviour of the machine learning predictor component. With this type of systems, it would be very difficult if not impossible to apply MATTER.

Construct validity threats concern the relationship between theory and observation. Concerning RQ1, they are due to how we measured the effectiveness of our approach in generating test suites able to detect bugs/faults. To minimise this threat, we decided to measure the effectiveness objectively, thanks to mutation testing. We adopted the Stryker mutator, a tool able to insert various types of bugs in the source code (and thus to mimic a relevant number of possible errors). Instead, for RQ2, construct validity threats are due to how we measured the time required to execute the generated test suites (both complete and minimised) on each IoT system. Also, in this case, the measure is objective since it was totally automated and, to minimise any fluctuation (due to possible active processes during the calculation), we averaged the obtained value three times. We have estimated that three times is sufficient

since we noticed that the variance is minimal. Another possible Construct validity threat is *Authors' Bias*. It concerns the involvement of the authors in manual activities conducted during the empirical study and the influence of the authors' expectations about the empirical study on such activities. To make our experimentation more realistic and to reduce as much as possible this threat, we adopted two IoT systems created in the context of previous works (SmartSantander, 2012; Istepanian et al., 2011; Parasoft, 2021; Sanchez et al., 2014) and thus with a behaviour specification already defined elsewhere. During the experimental part, we only modelled the behaviours of the chosen systems using UML State Machines and Class Diagrams as prescribed by our approach. Moreover, to reduce as much as possible any bias and render the evaluation more realistic in the evaluation of MATTER, we used Stryker and SQLMutation, two mutator generator tools, that independently are able to introduce bugs in the IoT SUT.

Threats to conclusion validity concern issues that may affect the ability to draw a correct conclusion, i.e. issues that may affect an adequate analysis of the data, as for example, using inadequate statistical methods. As our empirical study is similar to a case study and based on two IoT systems, we found it inappropriate to use statistical tests and therefore, this threat to validity does not apply to our case.

6 Related works

A common kind of model for describing behaviour that depends on sequences of events is the state machine. For this reason, in literature, there are several proposals deriving test cases from state machines (Bourhfir et al., 1997; Friedman et al., 2002; Hartman & Nagin, 2005; Utting, 2007).

Existing works cover different aspects of MBT, like test selection criteria (e.g. state coverage, transition coverage, or data-flow coverage) (Friedman et al., 2002), data coverage (e.g. pairwise or boundary analysis) (Bashir & Banuri, 2008; Edvardsson, 2002), prioritisation of relevant test cases (i.e. the ones that most likely will find bugs) (Gantait, 2011; Stallbaum et al., 2008), minimality of generated test suites (i.e. to generate the minimum number of test cases that will satisfy the desired coverage criteria) (Ammann & Offut, 2016) and transformation of abstract test cases in executable code (Utting, 2007). Many of the existing works do not cover the whole MBT process and generate only the list of test cases and input data required for testing, but do not produce executable test scripts as we did. This is the case of Bourhfir (Bourhfir et al., 1997) et al. that proposes an approach for generating test cases from an Extended Finite State Machine (EFSM) specification using the All Definition-Use Paths coverage criterion. To reduce the number of discarded test cases due to violated transition constraints, the executability of generated test cases is checked using symbolic execution. The final executable paths are evaluated symbolically in order to extract the corresponding input/output sequences which will be used during testing.

Friedman et al. (2002) describe several criteria for state machine coverage in test generation and a novel element based on testing constraints used to alleviate the state explosion problem and the test case explosion problem. MBT is known to easily generate a huge amount of tests from even small models (Grieskamp, 2006). Clearly, this is more a problem in practice than an advantage; in fact, this is the main concern mentioned by users of MBT

tools at Microsoft (Grieskamp, 2006). Friedman et al., moreover, describe also the test generation algorithms exploiting coverage criteria and testing constraints in the context of the tool they proposed, the GOTCHA tool (Hartman & Nagin, 1999).

Utting and Legeard, in their book 'Practical model-based testing' (Utting, 2007) describe several model-based testing techniques. They present techniques for test generation for finite state machines (simplifying, state machines without internal variables and guards on transitions), extended finite state machines (simplifying, state machines with internal variables and guards on transitions) and UML state machine diagrams. For simple finite state machines (FSM) (chapter 5), they use the Chinese postman algorithm (Thimbley, 2003) to generate the shortest path that visits every transition on the state machine graph. For extended finite state machines (EFSM) they present ModelJUnit⁵, a Java framework that allows to represent EFSM models and dynamically explore it. They also present methods to generate test cases from UML state machine diagrams. However, the description for UML state machines is not directly usable since the tool they used, LEIRIOS LTG/UML (Bernard et al., 2006), uses a proprietary constraint analysis engine to solve constraints imposed by guards. Utting and Legeard, in their book, also describe techniques to execute abstract test cases generated by model-based methods. In short, they propose three approaches: adaptation, transformation and mixed. In the adaptation approach, the software tester manually writes some code to wrap the SUT, exposing the interface required by the model. The transformation approach consists in translating abstract tests into executable test scripts, that can be expressed in a programming language like C or Java, or a test notation like TTCN-3. The mixed approach is a combination of the other two approaches. Our approach can be classified as mixed, with a strong prevalence of adaptation and just some transformation, used only to translate model operators (like assignment and relational operators) in the operators used by the target language.

The European research project AGEDIS (Automated Generation and Execution of Test Suites for DIstributed Component-based Software) (Hartman & Nagin, 2005) developed a suite of model-based tools that, starting from a model of the system made of UML class diagrams, object diagrams and state machine diagrams are able to produce test scripts for a distributed system. Prototypes of the tools have been tried in industrial settings (Craggs et al., 2003). UML models are translated in an intermediate format (IF), that represents the system as a collection of communicating processes, and each process behaviour is represented by states linked by guarded transitions. Starting from this model, the test generator produces an abstract test suite composed by paths (or directed graphs in case of non-deterministic systems) through the model, that are given to a test executor that runs them. The executor relies on test execution directives represented in XML to fill the gap between the model and the implementation details. In MATTER we used configuration files for a similar purpose.

Table 5 summarises the main similarities and differences among our proposal and the works of other researchers that are more closer to MATTER. Other MBT testing tools can be found in (Shafique & Labiche, 2015) where a systematic review about MBT tools relying on state-based models has been presented.

⁵ <https://sourceforge.net/projects/modeljunit/>

Table 5 Summary of the Similarities and Differences among our proposal and the related works

Paper	Similarities	Differences
Bourhfir (Bourhfir et al., 1997)	-MBT	-different output (test cases and not executable test scripts) -different domain (not specific for IoT systems) -different testing model (EFSM) -different coverage criterion (All Definition-Use Paths coverage criterion) -different approach to discard infeasible test cases (symbolic execution)
GOTCHA tool (Hartman & Nagin, 1999)	-MBT -similar output (executable test scripts)	-different testing model (FSM) -different domain (developed for hardware architecture models but used also for testing software systems) -different coverage criterion (state and transition coverage)
Utting and Legeard (Utting, 2007) (chapter 5)	-MBT -same testing model (UML state machines, but also FSM and EFSM) -similar output (executable test scripts)	-different domain (not specific for IoT systems) -different algorithms to generate test paths (e.g. Chinese postman algorithm (Thimbleby, 2003))
AGEDIS (Hartman & Nagin, 2005)	-MBT -similar testing model (UML class diagrams and state machine) -similar output (executable test scripts)	-differently from us the approach was applied in industrial settings -test scripts generation through intermediate format -different domain (distributed systems) -different approach to discard infeasible test cases (constraint analysis)

7 Conclusions and future work

In this paper, we have presented MATTER, a semi-automated model-based approach able to produce acceptance test suites for IoT systems. The approach is supported by a prototype tool that automatically generates entire test suites starting from an input model representing the expected behaviour of the system.

The effectiveness of the test suites generated by MATTER has been empirically evaluated using mutation testing. Over the two considered IoT systems, the test suites generated were able to detect, on average, the 91% of the mutants. We have also evaluated the time required to execute the generated test suite to understand the benefits of the novel minimisation process implemented in MATTER. The reduction is significant for both IoT systems, it ranges from 1/5 to 1/4 of the total time (with no negative effect on the bug-detection capability).

Our approach probably does not pay-off for small IoT systems like DiaMH, and partially also SmartParking, since its application requires a non-negligible effort. However, we

speculate that MATTER becomes cost-effective for complex IoT systems, when hundreds of test scripts must be generated. In these cases, indeed, a manual development of a test suite is a time-consuming task, and reaching manually a complete coverage of the model can be very hard (if not impossible). To better understand this aspect, we plan to extend our evaluation in several directions. First, we want to perform a controlled experiment to compare MATTER's automated approach with the manual one of generating test scripts. Second, we plan to evaluate the industrial applicability of our method thanks to the help of an industrial partner that develops real IoT systems in the smart city context (i.e. systems similar to SmartParking).

To improve the readability and creation of the models (in particular for complex IoT systems), we plan to extend MATTER to work with different state machines describing the system from different views: indeed, every state machine will focus on only one specific aspect of the system behaviour (e.g. main functionalities, reliability, connectivity). In this way, each state machine will describe the required behaviour from a specific point of view. Moreover, we plan to evaluate the effectiveness of the generated E2E test scripts in detecting different kinds of problems, beyond the possible bugs of the system source code: for instance, problem in the communication between the various components of the IoT system due to network errors, as we already investigated in the context of previous work (Leotta, 2019).

We also plan to investigate better the effect of adopting different coverage criterion in order to understand the trade-off between the complexity of the generated test suites and the bug detection capability.

Finally, we are preparing a documentation able to guide the MATTER's user on how to complete the various steps required (this will be fundamental for both the experiments with subjects and for proposing MATTER in the industry).

Funding Open access funding provided by Università degli Studi di Genova within the CRUI-CARE Agreement.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- SmartSantander working document 4. (2012). *Description of implemented IoT services*. <http://smartsantander.eu/downloads/Deliverables/D4.2.pdf>.
- ISO/IEC/IEEE. (2010). 24765:2010(E) *International Standard - Systems and Software Engineering – Vocabulary*. 1–418. <https://doi.org/10.1109/IEEESTD.2010.5733835>
- Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.
- Bashir, M. F., & Banuri, S. H. K. (2008). Automated model based software test data generation system. In *2008 4th International Conference on Emerging Technologies*, pp. 275–279. <https://doi.org/10.1109/ICET.2008.4777514>
- Bernard, E., Bouquet, F., Charbonnier, A., Legeard, B., Peureux, F., Utting, M., & Torrebore, E. (2006). *Model-based testing from UML models*, 94, pp. 223–230.

- Bourhfir, C., Dssouli, R., Aboulhamid, E., & Rico, N. (1997). *Automatic executable test case generation for extended finite state machine protocols* (pp. 75–90). US, Boston, MA: Springer. https://doi.org/10.1007/978-0-387-35198-8_6.
- Braiek, H. B., & Khomh, F. (2020). On testing machine learning programs. *Journal of Systems and Software*, 164, 110542. <https://doi.org/10.1016/j.jss.2020.110542>
- Briand, L. C., Labiche, Y., & Wang, Y. (2004). Using simulation to empirically investigate test coverage criteria based on statechart. In *Proceedings. 26th International Conference on Software Engineering*, pp. 86–95.
- Chekam, T. T., Papadakis, M., & Traon, Y. L. (2020). Muteria: An extensible and flexible multi-criteria software testing framework. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, pp. 97–100.
- Craggs, I., Sardis, E., & Heuillard, T. (2003). Agedis case studies: Model-based testing in industry. *Proceedings in 1st European Conference on Model Driven Software Engineering*, 192–132.
- Dustin, E., Rashka, J., & Paul, J. (1999). *Automated Software Testing: Introduction, Management, and Performance*. Inc, USA: Addison-Wesley Longman Publishing Co.
- Eck, M., Palomba, F., Castelluccio, M., & Bacchelli, A. (2019). Understanding flaky tests: the developer's perspective. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*. <https://doi.org/10.1145/3338906.3338945>
- Edvardsson, J. (2002). *A survey on automatic test data generation*.
- Escobar-Velázquez, C., Riveros, D., & Linares-Vásquez, M. (2020). Mutapk 2.0: a tool for reducing mutation testing effort of android apps. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1611–1615.
- Friedman, G., Hartman, A., Nagin, K., & Shiran, T. (2002). Projected state machine coverage for software testing. *SIGSOFT Software Engineering Notes*, 27(4), 134–143. <https://doi.org/10.1145/566171.566192>.
- Gantait, A. (2011). Test case generation and prioritization from UML models. In *Proceedings of 2nd International Conference on Emerging Applications of Information Technology*, pp. 345–350. *EAIT 2011*, pp. 345–350.
- Grieskamp, W. (2006). Multi-paradigmatic model-based testing. pp. 1–19. <https://doi.org/10.1007/119401971>
- Grün, B. J. M., Schuler, D., & Zeller, A. (2009). The impact of equivalent mutants. In *Proceedings of 2nd International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009*, pp. 192–199. <https://doi.org/10.1109/ICSTW.2009.37>
- Hartman, A., & Nagin, K. (1999). Tbeans, software test toolkit. *Proceedings of the 12th International Software Quality Week (QW 1999)*, 445, 450.
- Hartman, A., & Nagin, K. (2005). *The agedis tools for model based testing*. In *UML Modeling Languages and Applications (Berlin, Heidelberg, 2005)*, Jardim Nunes, N., Selic, B., Rodrigues da Silva, A., & Toval Alvarez, A., Eds. Springer Berlin Heidelberg, pp. 277–280.
- Istepanian, R. S. H., Hu, S., Philip, N. Y., & Sungoor, A. (2011). The potential of Internet of m-health Things m-IoT for non-invasive glucose level sensing. In *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pp. 5264–5266. <https://doi.org/10.1109/TEMBS.2011.6091302>
- Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5), 649–678. <https://doi.org/10.1109/TSE.2010.62>.
- Kaminski, G., Praphamontipong, U., Ammann, P., & Offutt, J. (2011). A logic mutation approach to selective mutation for programs and queries. *Information and Software Technology*, 53(10), 1137–1152.
- Korel, B. (1990). Automated software test data generation. *IEEE Transaction on Software Engineer*, 16(8), 870–879. <https://doi.org/10.1109/32.57624>.
- Leotta, M., Cerioli, M., Olinas, D., & Ricca, F. (2020). Two experiments for evaluating the impact of Hamcrest and AssertJ on assertion development. *Software Quality Journal (SQJ)*, 28, 1113–1145. <https://doi.org/10.1007/s11219-020-09507-0>.
- Leotta, M., Clerissi, D., Olinas, D., Franceschini, L., Ancona, D., Ricca, F., & Ribaudo, M. (2019). Comparing testing and runtime verification of IoT systems: A preliminary evaluation based on a case study. In *Proceedings of 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, SciTePress Digital Library, pp. 434–441. <https://doi.org/10.5220/0007745604340441>
- Leotta, M., Clerissi, D., Olinas, D., Ricca, F., Ancona, D., Delzanno, G., Franceschini, L., Ribaudo, M. (2018). An acceptance testing approach for Internet of Things systems. *IET Software*, 12, 430–436. <https://doi.org/10.1049/iet-sen.2017.0344>.
- Leotta, M., Clerissi, D., Ricca, F., & Tonella, P. (2016). Approaches and tools for automated End-to-End web testing. *Advances in Computers*. Elsevier, Volume 101, pp. 193 – 237. <https://doi.org/10.1016/b978-0-12-405111-0.00011-0>
- Leotta, M., Ricca, F., & Tonella, P. (2021). SIDEREAL: Statistical adaptive generation of robust locators for Web testing. *Journal of Software: Testing, Verification and Reliability (STVR)*, 31. <https://doi.org/10.1002/stvr.1767>

- McMinn, P. (2011). Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 153–163. <https://doi.org/10.1109/ICSTW.2011.100>
- Mesbah, A., & van Deursen, A. (2009). Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society*, pp. 210–220. <https://doi.org/10.1109/ICSE.2009.5070522>
- Myers, G. J., Sandler, C., & Badgett, T. (2012). The art of software testing, 3rd ed. John Wiley & Sons, Hoboken and N.J.
- Offutt, A. J., & Untch, R. H. (2000). Mutation 2000: Uniting the Orthogonal. *Springer, 2001*, 34–44. https://doi.org/10.1007/978-1-4757-5939-6_7.
- Olianas, D., Leotta, M., & Ricca, F. (2020). An Approach and a Prototype Tool for Generating Executable IoT System Test Cases. In *Quality of Information and Communications Technology (Cham, 2020)*, Shepperd, M., Brito e Abreu, F., Rodrigues da Silva, A., & Pérez-Castillo, R., Eds. Springer International Publishing, pp. 383–398.
- Parasoft. (2017). End-to-end testing for IoT integrity. *Technical Reports*. <https://alm.parasoft.com/end-to-end-testing-for-iot-integrity>.
- Reggio, G., Leotta, M., Cerioli, M., Spalazzese, R., & Alkhabbas, F. (2020). What are IoT systems for real? an experts survey on software engineering aspects. *Internet of Things 12*, 100313. <https://doi.org/10.1016/j.iot.2020.100313>
- Ricca, F., Torchiano, M., Leotta, M., Tiso, A., Guerrini, G., & Reggio, G. (2018). On the impact of state-based model-driven development on maintainability: A family of experiments using UniMod. *Journal of Empirical Software Engineering (EMSE)*, 23(3), 1743–1790.
- Samek, M. (2008). *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. <https://doi.org/10.1007/s10664-017-9563-8>
- Sanchez, L., Munoz, L., Galache, J. A., Sotres, P., Santana, J. R., Gutierrez, V., Ramdhany, R., Gluhak, A., Krco, S., Theodoridis, E., & Pfisterer, D. (2014). SmartSantander: IoT experimentation over a smart city testbed. *Computer Networks*, 61, 217–238. Special issue on Future Internet Testbeds Part I. <https://doi.org/10.1016/j.bjp.2013.12.020>
- Shafique, M., & Labiche, Y. (2015). A systematic review of state-based test tools. *International Journal on Software Tools and Technology Transfer*, 17(1), 59–76. <https://doi.org/10.1007/s10009-013-0291-0>.
- Shalev-Shwartz, S., et al. (2011). Online learning and online convex optimization. *Foundations and trends in Machine Learning*, 4(2), 107–194.
- Stallbaum, H., Metzger, A., & Pohl, K. An automated technique for risk-based test case generation and prioritization. In *Proceedings of 3rd International Workshop on Automation of Software Test (2008), AST 2008*, pp. 67–70.
- Thimbley, H. (2003). The directed chinese postman problem *Software: Practice and Experience*, 33(11), 1081–1096. <https://doi.org/10.1002/spe.540>.
- Tracey, N., Clark, J., Mander, K., & McDermid, J. (1998). An automated framework for structural test-data generation. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering, ASE 1998, IEEE*, pp. 285–288.
- Tuya, J., Dolado, J., Suarez-Cabal, M. J., & de la Riva, C. (2008). A controlled experiment on white-box database testing. *ACM SIGSOFT Software Engineering Notes*, 33(1), 1–6. <https://doi.org/10.1109/MUTATION.2006.13>.
- Tuya, J., Suarez-Cabal, M. J., & de la Riva, C. (2006). Sqlmutation: A tool to generate mutants of sql database queries. In *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, pp. 1–1.
- Tuya, J., Suarez-Cabal, M. J., & de la Riva, C. (2007). Mutating database queries. *Information and Software Technology*, 49(4), 398–417. <https://doi.org/10.1016/j.infsof.2006.06.009>.
- Utting, M. (2007). and Legeard, B. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc.
- Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification & Reliability*, 22(5), 297–312. <https://doi.org/10.1002/stvr.456>.
- Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., & Wessln, A. (2012). *Experimentation in Software Engineering*. Incorporated: Springer Publishing Company.
- Yen, J. Y. (1971). Finding the k shortest loopless paths in a network. *Management Science*, 17(11), 712–716.

Dario Olianias is a PhD student at the University of Genova, Italy. He received his Master degree in Computer Science in 2019 with a thesis on automated testing of IoT systems. He is author or coauthor of some papers published in international journals and conferences/ workshops.

Maurizio Leotta is a researcher at the University of Genova, Italy. He received his PhD degree in Computer Science from the same University, in 2015, with the thesis “Automated Web Testing: Analysis and Maintenance Effort Reduction”. He is author or coauthor of more than 80 research papers published in international journals and conferences/workshops. His current research interests are in software engineering, with a particular focus on the following themes: Web, Mobile, and IoT application testing, functional test automation, empirical software engineering, business process modelling and model-driven software engineering.

Filippo Ricca is an associate professor at the University of Genova, Italy. He received his PhD degree in Computer Science from the same University, in 2003, with the thesis “Analysis, Testing and Re-structuring of Web Applications”. In 2011 he was awarded the ICSE 2001 MIP (Most Influential Paper) award, for his paper: “Analysis and Testing of Web Applications”. He is author or coauthor of more than 100 research papers published in international journals and conferences/ workshops. Filippo Ricca was Program Chair of CSMR/WCRE 2014, CSMR 2013, ICPC 2011, and WSE 2008. His current research interests include: Software modeling, Reverse engineering, Empirical studies in Software Engineering, Web applications and Software Testing.