Towards an Approach for Developing and Testing Node-RED IoT Systems

Diego Clerissi
Dip. di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
diego.clerissi@dibris.unige.it

Maurizio Leotta
Dip. di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
maurizio.leotta@unige.it

Gianna Reggio
Dip. di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
gianna.reggio@unige.it

Filippo Ricca
Dip. di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
filippo.ricca@unige.it

ABSTRACT

Node-RED is a visual tool based on the flow-based programming paradigm and built on NodeJS, which is used for developing IoT systems. In Node-RED, the developer can follow her own personal flavour for wiring devices and online services together, and the same system can be developed in many different ways. Each day, the Node-RED community submits to users novel solutions, and even if there exist frameworks for testing Node-RED flows, they are not supported by a systematic testing technique. Hence, the freedom granted by Node-RED may hinder the understandability of the produced artefacts and the detection of faults.

In this work, we propose a preliminary version of an approach for developing and testing a Node-RED system starting from a UML model of its dynamic and static aspects. A JSON object representing the Node-RED system is generated from the model, while executable Javascript test scripts relying on the Mocha test framework are generated from selected portions of the model, enriched with control points to perform checks over the system properties. We believe that a model produced with our approach may help in the early system validation by detecting faults and deviations from its expected behaviour.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**; **Software verification and validation**; • **Computer systems organization → Distributed architectures**;

KEYWORDS

Node-RED, UML, Modelling, Testing, IoT, Javascript, Mocha

ACM Reference Format:
Diego Clerissi, Maurizio Leotta, Gianna Reggio, and Filippo Ricca. 2018. Towards an Approach for Developing and Testing Node-RED IoT Systems.

EnSEmble '18, November 4, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-6054-8/18/11…$15.00
https://doi.org/10.1145/3281022.3281023

In *Proceedings of the 1st ACM SIGSOFT International Workshop on Ensemble-Based Software Engineering (EnSEmble '18), November 4, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3281022.3281023

1 BACKGROUND

In the context of the Internet of Things (IoT), where interconnected and heterogeneous devices cooperate to complete tasks, sometimes complex and safety-critical, proposing effective methods and approaches for developing and testing IoT systems is essential, but brings a number of challenges [1].

Node-RED is a visual programming tool inspired by the flow-based programming paradigm [7] and built on the NodeJS framework, which provides solutions for developing IoT systems in terms of nodes and flows. In Node-RED, a *node* represents the logics of a device or, more generally, of a service provided by a system. Nodes are wired together any time they have to cooperate/communicate in order to complete tasks, hence they compose *flows*, which are the portions of a system that logically group together sequences of functionalities. The community behind Node-RED is quite active, and new nodes based on the emerging technology are submitted daily. There exist nodes for completing a variety of tasks, like reading values from a database, implementing a Javascript function, receiving the feeds from a Twitter account, establishing a communication between two devices using the MQTT protocol, and more.

Like any other programming language or tool, Node-RED gives the developer the freedom to follow her own personal flavour for developing a system. For example, she may develop it using a unique flow employing function nodes any time a global variable has to be initialized, or she may choose to separate the logics of the system in different flows and use more fashionable nodes for the initializations (e.g., change node[1]). This freedom, however, has the negative effect that the developer may produce a working but unintelligible system, hindering subtle faults and deviations from the system expected behaviour that could be hard to detect and fix.

Indeed, understanding and testing Node-RED systems can be difficult. An example is a flow having three nodes sequentially wired: an "inject node" to simulate an external event, like clicking on a physical button, followed by a "function node" to filter out the

[1] https://nodered.org/docs/user-guide/nodes#change

bad values received from the external source, followed by a "debug node" to display the good ones. This simple example can be directly tested in Node-RED by checking the values displayed by the debug node, or by using one among the nodes purposely provided by the Node-RED community for the verification of nodes and flows (e.g., assert node[2]). However, in case the example becomes trickier, it is undeniable that, due to the heterogeneity of the involved functionalities that in some cases may employ freshly released and barely tested nodes, testing a complete flow can be hard. The Node-RED community has tried to answer to the users' testing demand[3], and even if finding online materials is relatively easy, like guides[4] and specific posts on message boards[5], neither systematic approaches nor methods have been yet proposed to test Node-RED flows.

In recent years, some proposals for assuring the quality of IoT systems [4, 6] and, more generally, of Cyber-Physical Systems (CPSs) [3, 12] have emerged, but these works do not specifically aim at supporting the user in the development and testing activities of a Node-RED system and do not provide solutions for automatically generating Node-RED flows and test scripts from a model.

Concerning monitoring and formal verification of CPSs, Kane et al. [3] presented a runtime monitor verification technique to describe and detect properties violations of safety-critical systems, formally described. Their research challenges are different from ours and concern, for instance, how to properly abstract a system based on a limited perspective of its internal behaviour and how such abstraction is close enough to the real system, whereas in our case the system behaviour has to be defined to guide the development and the testing activities.

In a previous work [6], we proposed an approach for testing IoT systems developed in Node-RED and equipped with a User Interface (UI). The approach requires to model the system behaviour as a UML state machine and to manually extract some test scenarios making assertions over changes in the UI, hence it is specifically focused on UI testing and does not aim at automatically generating a Node-RED system from a model.

Kim et. al [4] introduced a service-based framework for testing IoT systems, by adapting and evolving traditional testing methodologies to the context of IoT. The goals of the paper are different from ours and do not directly answer problems concerning IoT systems development, in particular in Node-RED.

Therefore, corroborated by the aforementioned reasons and inspired by the Node-RED testing framework[6], which is now a practical solution for testing Node-RED flows from a unit level, in this paper we propose an idea that should answer some of the emerged problems and lead to an approach for effectively developing and testing Node-RED systems. The preliminary version of the approach, outlined in the paper, is based on the authors' experience [5, 6] and on some of the leading guidelines and ingredients of the modelling methods proposed by the authors' [2, 8, 9]. First, the dynamic and static aspects of the system are modelled using UML activity and class diagrams, from which JSON objects representing the working Node-RED flows compliant to the UML model can be generated. Then, by selecting portions of the model enriched with control points to check over the system properties, it is possible to generate test scripts able to exercise the corresponding system flows. The test scripts will be executed using Mocha[7], a flexible and practical test framework that runs on NodeJS and supports many assertion libraries. Interactions in the early phase of the approach between the professionals and the stakeholders are necessary, in order to obtain useful feedbacks for developing the right system without introducing faults or deviations from the system expected behaviour and to focus on the stakeholders' needs.

In Section 2 we introduce the running example used for presenting our approach, which is outlined in Section 3, while conclusions and future work are given in Section 4.

2 RUNNING EXAMPLE

To present our approach, as running example we have chosen a simple IoT system to be developed in Node-RED and consequently tested. The system is composed of a `Reader Device` and a `Temperature Setter`. The `Reader Device` reads from a continuous file stream the last 24 hours environmental degrees Celsius temperatures (range [-20, 40]) recorded outside a room by an external source. The values are transmitted to the `Temperature Setter`, by using the MQTT protocol, which evaluates its internal state in the following way. It computes the average of the received values and checks it against the one computed the day before and, depending on the variation between the two averages, it sets its internal state to `Colder Temperature`, if the temperature inside the room has to be increased (i.e., the old average is higher than the new one), to `Same Temperature` if no change is needed, or to `Warmer Temperature`, if the temperature inside the room has to be reduced. Depending on the internal state and on other parameters, finally the `Temperature Setter` sets the daily temperature inside the room.

We have considered the possibility that the developers may not have decided yet how to precisely handle each `Temperature Setter` state; for example, they may want it to be implemented in Node-RED as a variant of the many flows involving a thermostat node[8]. The development and the testing activities over such system should not be limited because of some unclear parts in its behaviour, even more if the system is complex or safety-critical; instead, the professionals may want to have a portion immediately working as compliant, even if incomplete, and another portion to be iteratively refined and tested in the future. A sketched representation of the behaviour of the running example is shown in Figure 1. Notice that step `5. Set Temperature*` is labelled with an asterisk to indicate that still has to be precisely defined.

3 THE APPROACH

The approach we propose in this paper, sketched in the activity diagram of Figure 2, is based on our personal experience in Node-RED [5, 6] and on the issues and the open questions that are daily posted by the users community.

As shown in Figure 2, different tasks have to be completed in order to generate the Node-RED flows compliant to the system

[2] https://www.npmjs.com/package/node-red-contrib-assert
[3] https://github.com/node-red/node-red/wiki/Testing
[4] e.g., http://noderedguide.com/
[5] e.g., https://discourse.nodered.org/
[6] https://www.npmjs.com/package/node-red-node-test-helper
[7] https://mochajs.org
[8] e.g., https://www.npmjs.com/package/node-red-contrib-ramp-thermostat

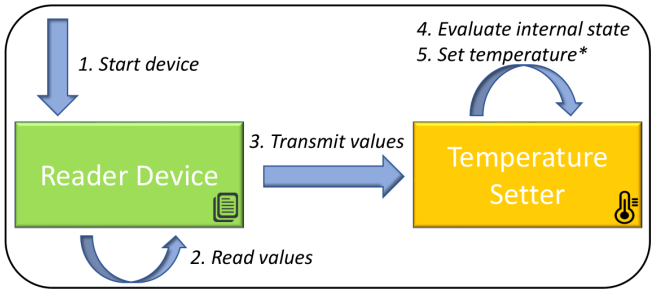

Figure 1: The running example.

expected behaviour and the executable test scripts to check if faults or deviations have been introduced during the system development. "Node-RED Flows Generation" and "Mocha Test Script Generation" tasks are marked in red because they will be tool-supported.

To represent the behaviour of Node-RED IoT systems we have chosen UML, since is widely known and used [10, 11] and can naturally describe the dynamic aspects of Node-RED flows, by means of activity diagrams, and the static properties of the nodes (e.g., the body of a function, the TCP communication settings), by means of class diagrams and OCL expressions. Moreover, the XML Metadata Interchange (XMI) standard adopted by UML models is supported by many tools (e.g., Papyrus[9]) and transformations to other languages already exist (e.g., ecorejs[10] for Javascript).

3.1 Behaviour Modelling

This task requires the designer to model, with a UML activity diagram, the behaviour of the system that is intended to be developed and tested. In this task, only the nodes and the wires between them are important. Currently, we have restricted the activity diagram to the following UML constructs that, in our opinion, are sufficient to represent the basic Node-RED nodes[11]: *action nodes, decision/merge nodes, fork/join nodes, object nodes, swimlanes, activity edge connectors, initial nodes, activity final nodes, flow final nodes, send signal events, accept (time) events, exception handlers, input pins, and control/object flows.*

In our approach, each Node-RED node has its own UML counterpart. For example, the inject node transmits information based on the external event it receives, which can be repeated in time (e.g., send a message every 10 seconds), hence we have chosen to represent it as a UML accept (time) event, where the event can be timed depending on its repeatability. Another example is the switch node, that in UML is represented with a decision node. To improve the model understandability, stereotypes representing the various nodes have to be added to the corresponding UML constructs.

Messages passing is an activity performed by almost every Node-RED node, but in some cases the message a node returns is an untouched or a slight changed version of the received one. Hence, while modelling the behaviour of a system, only when needed to improve the understandability, we have chosen to represent messages as UML object nodes exposing their properties in the

[9]https://www.eclipse.org/papyrus/
[10]http://emfjson.org/projects/ecorejs/latest/
[11]http://noderedguide.com/node-red-lecture-3-basic-nodes-and-flows/

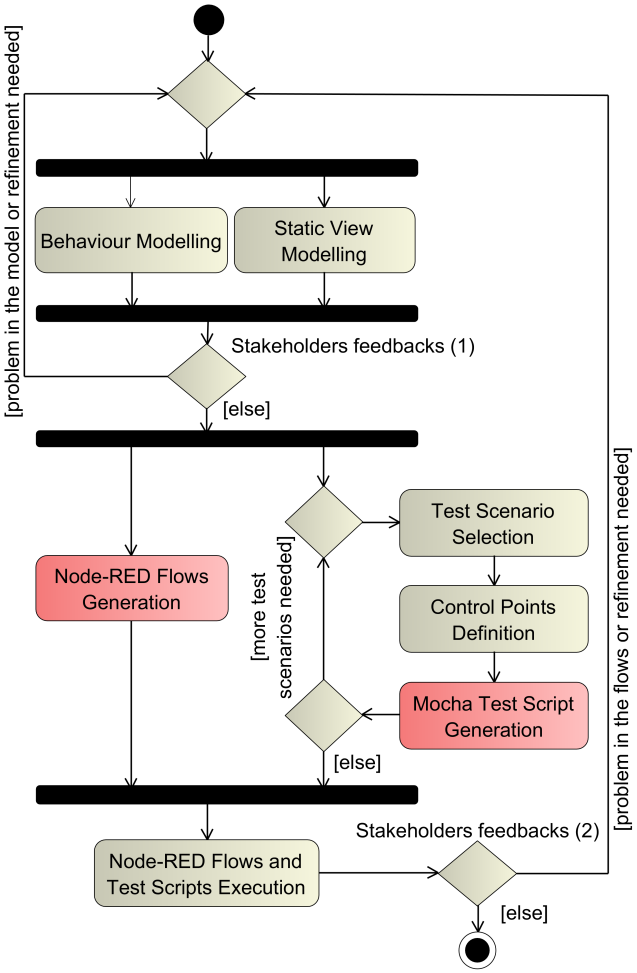

Figure 2: The proposed approach.

form of `{property1: value1, . . . , propertyN: valueN}`. For example, if a function node adds a property `P` with value `V` to a received message, the returned message will be modelled as a UML object node labelled with `{..., P:V, ...}`.

We have decided to use swimlanes for representing the main Node-RED flows of a system. Each lane corresponds exactly to a main flow, then the placement of any UML construct representing a Node-RED node within a certain lane determines the node scope to the corresponding flow. In our running example of Figure 1, the system is composed of two devices, hence it will require two lanes representing its two main flows.

At this stage, modelling IoT systems in Node-RED can be tough, due to the number of heterogeneous and interconnected devices to handle and to all the technical/configuration details required by Node-RED nodes. For this reason, our approach introduces the concept of mocked portions of a system behaviour. A *mocked portion* is something that the designer may not want to model yet, because of not immediate interest or because further time for thinking is required (in Figure 1, see step `5. Set temperature*`), and then she

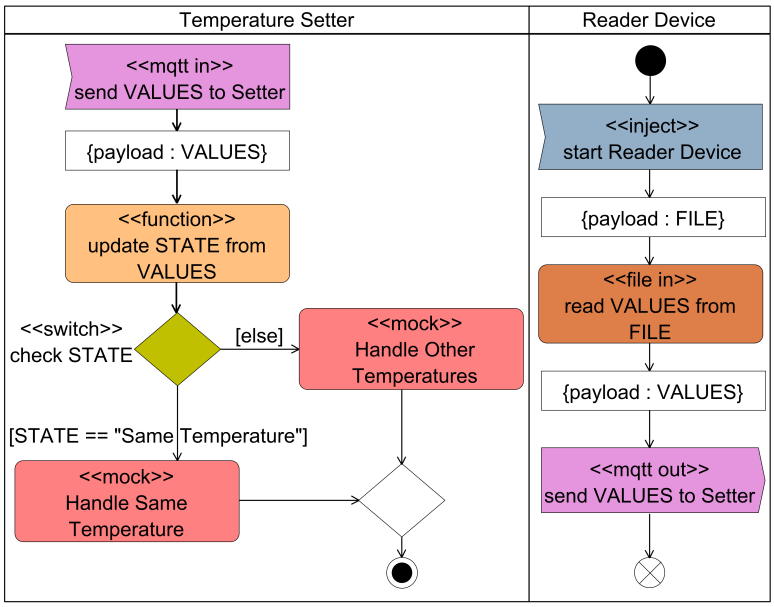

Figure 3: The behaviour of the running example.

mocks it with a scripted behaviour. A UML activity construct stereotyped by ≪mock≫ is used any time a portion of the behaviour of a system has to be mocked.

Figure 3 provides a simplified UML activity diagram representing the behaviour of our running example. Two lanes are used to delimit the system components: a `Temperature Setter` and a `Reader Device`. The system starts once the `Reader Device` receives an external event, modelled as an accept event and stereotyped with ≪inject≫. The node returns a message (a UML object node) having as payload the name of the file stream where the temperatures are stored (the variable `FILE`). Then, a ≪file in≫ node modelled as a UML action node reads the values from the file and returns them, again as a message payload set to `VALUES` variable, to the `Temperature Setter` through a ≪mqtt out≫ node, modelled as a UML send signal node, and the flow ends, as shown by the UML flow final node. The `Temperature Setter` receives the message using a ≪mqtt in≫ node named accordingly and returns it to a ≪function≫ node, which updates the internal state (a flow variable named `STATE`) depending on the received `VALUES`; no details about the function have to be given in the activity diagram, since they will be provided in the static view (see Figure 2). Then, a ≪switch≫ node receives the message from the function node and checks the value of `STATE`, resulting in two possible activities: `Handle Same Temperature` and `Handle Other Temperatures`. These two activities are modelled as mocked portions of the system, as shown by their stereotypes, which means that they are not yet intended to be developed and will present a scripted behaviour, specified in the static view, without blocking the execution or the testing of the system.

From the example, it is notable that not all the UML constructs correspond to Node-RED nodes; indeed, the constructs with neither colours nor stereotypes are just used for modelling purposes. For example, the UML flow final node at the end of the `Reader Device` lane states when the flow ends, but has no equivalent representation in Node-RED. Similarly, the UML merge node of the `Temperature Setter` lane is used only for merging together the two alternatives exiting from the previous UML decision node. More generally, there is not a bijective correspondence between the UML constructs in the activity diagram and the nodes in the Node-RED flows; in fact, the two notations present different syntax and semantics which clearly have to be deeply investigated. In any case, we think that UML includes all the information needed to generate Node-RED flows from a UML model.

3.2 Static View Modelling

This task is conducted in parallel with the "Behaviour Modelling" task, see Figure 2, and requires the designers to model, with a UML class diagram, the static view of the system that is intended to be developed and tested.

More specifically, the class diagram exposes classes, named *flow classes*, each one representing a lane of the activity diagram (i.e., the main Node-RED flows of the system). A flow class contains an operation for each UML construct representing a Node-RED node included in the corresponding lane. Finally, OCL notes are attached to each flow class to define their operations, i.e., the properties of the nodes, formulated as a conjunction of `Property = Value`. The definition of the operations does not require parameters or returned values. Indeed, most of Node-RED nodes receive messages and return messages, sometimes changing their structures, hence making messages passing explicit would not add any information; instead, whenever a message has to be made explicit, it is modelled in the activity diagram as a UML object node exposing all the interesting properties, as shown in Figure 3.

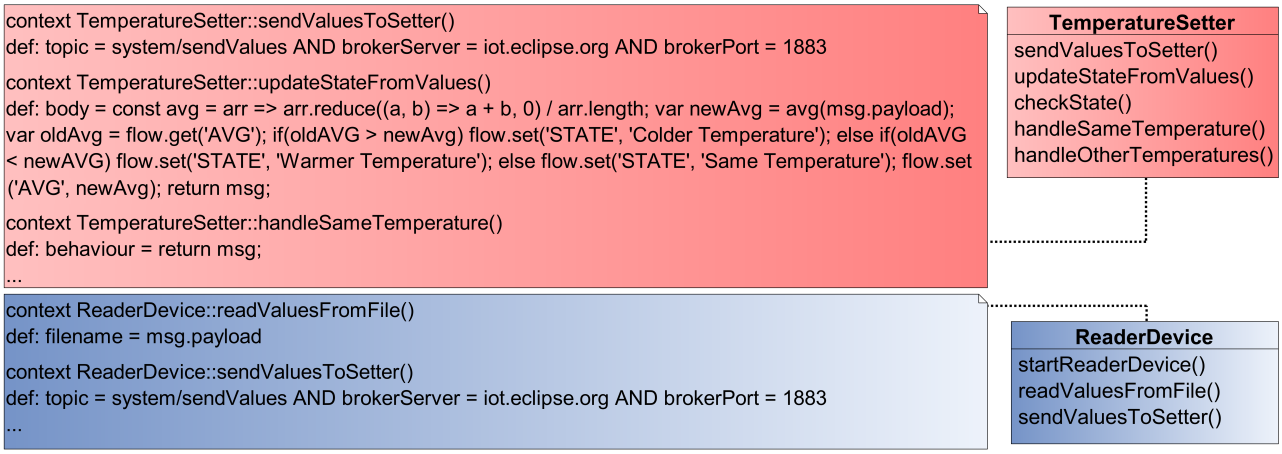

Figure 4: A portion of the static view of the running example.

Our approach requires iterative steps for modelling the behaviour and the static view of a system, see the loops in the process of Figure 2, hence the definition of some nodes properties which may result too complex at this early stage can be postponed, for instance those requiring technical details (e.g., the server name and the port number of a communication node) or programming skills (e.g., the Javascript body of a function node). A partial view of the class diagram, defining some properties of the nodes of the activity diagram modelled in Figure 3, is shown in Figure 4.

Since the activity diagram is composed of two lanes, corresponding to the `Reader Device` and the `Temperature Setter` main flows of the system, the class diagram presents two flow classes, each one having an attached OCL note defining its operations. For instance, `sendValuesToSetter` of the `TemperatureSetter` class represents the homonym MQTT node receiving the values from the `Reader Device`, hence it requires properties such as the `topic` (i.e., basically, the message identifier), the `broker's server` and the `broker's port`. Instead, `updateStateFromValues` of the same class represents the homonym function node and has a property named body which embodies its logics in Javascript language, i.e., *it computes the average of the received values and compares it with the one computed the day before, stored in a flow variable named* `AVG`. *Depending on which average is higher, it changes a flow variable named* `STATE` *that will be used for the next temperature setting.* Notice also the definition of `handleSameTemperature`; the operation represents the homonym mocked portion of the system and its behaviour is defined to simply returning the message it receives, hence doing nothing. This definition will have to be changed once that mocked portion of the system is clearer.

3.3 Stakeholders Feedbacks

Our approach requires strong interactions between the stakeholders and the professional figures responsible for modelling, developing and testing the system, as shown in Figure 2. Indeed, since having Node-RED flows aligned with the system expected behaviour is a mandatory requirement of our approach, it is imperative that the stakeholders can analyse the produced artefacts and provide feedbacks. This can happen in two phases: one at the beginning of the process (`Stakeholders feedbacks (1)`), when a problem in the model is identified or a refinement of the model is needed (e.g., update the class diagram by defining a new node property or update the activity diagram by modifying a flow), and one at the end of the process (`Stakeholders feedbacks (2)`), when a problem in the generated Node-RED flows or in the test scripts is identified or when a refinement of the model is needed (e.g., some mocked portions have to be defined or a node property has to be fixed).

3.4 Node-RED Flows Generation

Once the system has been modelled, it is possible to transform the produced UML model into Node-RED, by generating the flows and the nodes properties from the activity and the class diagrams, respectively. Basically, it is a transformation from XMI (for the UML perspective) to JSON (for the Node-RED perspective), which is intended to be automated. The transformation will apply a procedure, sketched as follows, which will require a fine-tuning after a proper application on real case studies:

- For each lane L_i in the activity diagram, generate an empty Node-RED flow F_i;
- For each UML construct C_k in lane L_i, if C_k is stereotyped as S_k, generate a Node-RED node N_k in flow F_i having the form $\{id : n_k, name : C_k, type : S_k, z : L_i, wires : [\,]\}$, where z is the property that Node-RED uses to identify a flow;
- For each flow class FC_i in the class diagram, for each operation O_{ki}, add the definition of O_{ki}, having the form $property_1 = value_1, \ldots, property_N = value_N$ to node N_k in flow F_i, changing = with :;
- For each couple of UML constructs C_n and C_m in lane L_i, having stereotypes S_n and S_m respectively, if there exist a sequence of transitions from C_n to C_m such that no other stereotyped UML construct is in the sequence, and if S_n permits output wires and S_m permits input wires, then update property *wires* of node N_n in flow F_i from $[\ldots]$ to $[\ldots, [n_m]]$.

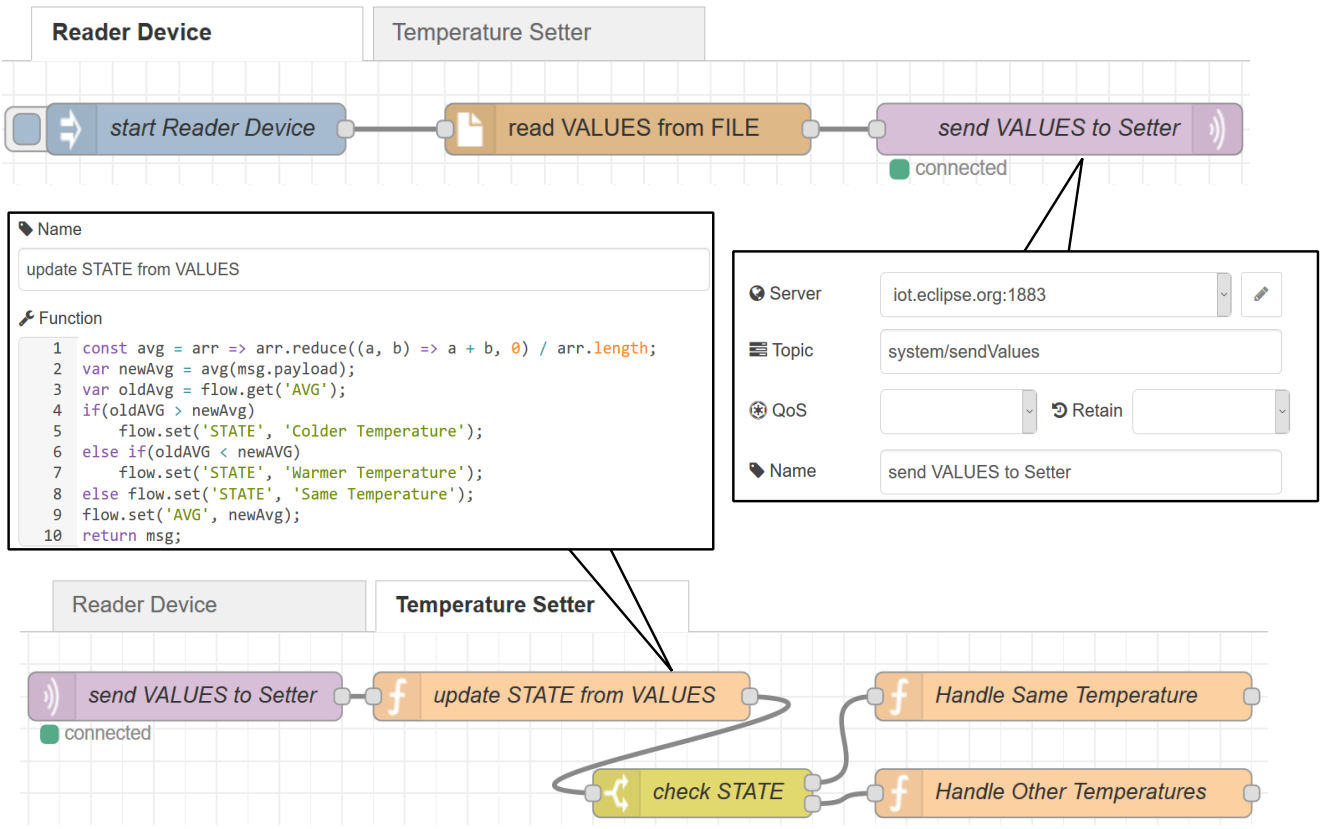

Figure 5: The JSON flows and some nodes properties of the running example, as seen in Node-RED.

A possible outcome of the procedure is given in Figure 5, where the Node-RED flows of the `Reader Device` and the `Temperature Setter` components of the running example are generated from the activity and the class diagrams shown in Figures 3 and 4. Notice the mocked portions transformed into function nodes, each one having a scripted behaviour.

3.5 Test Scenarios Selection

In our approach, the testing activity over the system is performed in parallel with the generation of the Node-RED flows, as shown in Figure 2. The produced activity diagram describes the complete system behaviour, including mocked portions, therefore test scenarios can be selected from it.

We define a *test scenario* as a physically or a logically connected portion of the system behaviour, composed of different UML constructs. The connection between UML constructs is essential for having a test scenario: it is physical when two UML constructs `C1` and `C2` are directly connected with a UML control/object flow; it is logical when `C1` and `C2` logically represent a sequence of possible events, for instance if `C1` is a UML send signal node stereotyped with ≪mqtt out≫ and `C2` is a UML accept event node stereotyped with ≪mqtt in≫.

Since a test scenario is partial it does not represent a fully working Node-RED flow, hence it may lack preceding steps where variables and message properties are set. Then, each test scenario has to be preceded by a *tailored mocked portion*, responsible for setting the used global and flow variables and the message properties. As the name suggests, each tailored mocked portion is tailored to a specific test scenario. For instance, we may want to check that a component `A` can send a message `M` to a component `B` and, based on the payload of `M`, `B` can execute either sub-flows `S1`, `S2` or `S3`; hence, depending on the way the payload is set by the tailored mocked portion defined for that test scenario, one among those three sub-flows may be exercised.

As for the mocked portions introduced while modelling the behaviour of a system, even the tailored mocked portions of test scenarios are represented as UML activity constructs stereotyped with ≪mock≫, in this case preceding the first UML constructs of the test scenarios they are associated with and placed in special lanes (e.g., Testing lane). Moreover, each tailored mocked portion has to be defined by adding to the class diagram used for modelling the static view of the system a new class named as the newly introduced lane, that will represent the tailored mocked portion as an operation and will define it in an OCL note in the form of `behaviour = B`, where `B` is expressed in Javascript.

At this stage of the work we have not yet defined a precise strategy for selecting the best test scenarios in terms of system coverage and the smartest way for customizing the tailored mocked portions. We intend of course to investigate on these two topics, in particular in inferring the variables used by test scenarios and in generating proper input data.

3.6 Control Points Definition

To proceed in the testing activity of the system, the selected test scenarios must be completed by adding some control points. In our approach, a *control point* corresponds to any assertion formulated using a testing framework for a specific programming language (e.g., JUnit for Java), hence defines a check over a system property, i.e., a global/flow variable or a message property, and is represented as a UML action node stereotyped with ≪control point≫. The idea of adding control points within Node-RED flows has been inspired by some of the Node-RED nodes and frameworks having verification purposes, e.g., the `node-red-contrib-assert` node and the `node-red-node-test-helper` framework mentioned in Section 1, which can be added within a Node-RED flow to intercept the information passed among the observed nodes.

Each control point attached to a test scenario has to be added to the same lane of the tailored mocked portion for that test scenario and is definition is given in an OCL note attached to the class in the class diagram corresponding to that lane, in the form of `check = C`, where `C` is expressed in Javascript.

We still have to think about all the possible checks to be defined by control points, but theoretically they could be formulated relying on the plethora of libraries and modules running in Javascript and NodeJS (e.g., Should[12], Chai[13], Assert[14]). Examples of checks using the Should library are `var.should.be.equal(V)`, where `var` is a global/flow variable or a message property and `V` is a primitive value, or `msg.should.have.property(P, V)`, where `P` is the name of a property that a message `msg` should have and `V` is a primitive value. More complex checks over system properties will be investigated, e.g., checks over the time taken by a communication, comparisons of the output produced by multiple executions of the same test scenario, and so on.

Figure 6 shows, on top, a test scenario selected from the system behaviour modelled in Figure 3. Since the test scenario focuses on just a portion of the `Temperature Setter` lane, a tailored mocked portion named `Mock Reader Device` is introduced at the beginning of the test scenario and added to a new lane named `Testing`. The test scenario ends after the function node, ignoring what happens next, therefore a control point is attached to the transition exiting from it. On the bottom of Figure 6, the class representing the `Testing` lane is shown, including the note defining both the tailored mocked portion and the control point. The tailored mocked portion is defined by feeding the test scenario with a message payload of three temperatures (27.5, 26, and 24.5), to simulate the values the `Reader Device` reads from the file stream, and by setting a flow variable named `AVG` to 22. This means that the average temperature computed by the function node of the `Temperature Setter` (see its definition in Figures 4 and 5) over the three received values is 26, slightly higher than the one computed the day before and stored in `AVG`, hence the value of the `STATE` variable after the function should be equal to `Warmer Temperature`, as required in the definition of the control point named `check STATE`.

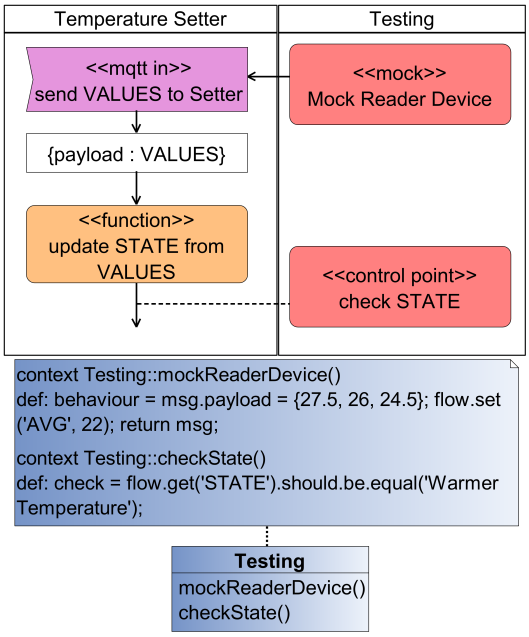

Figure 6: A test scenario (top) and the definition of the tailored mocked portion and control point (bottom) of the running example.

3.7 Mocha Test Scripts Generation

Once the control points are introduced in a test scenario and consequently defined, it is possible to generate the corresponding test script, relying on the Mocha test framework. The generation of the test scripts is intended to be automatically conducted by a tool that will apply the following sketched procedure. Given a test scenario TS:

- Generate an empty Mocha test script MTS;
- Generate the JSON flow F from TS and declare it in MTS;
- Extract the unique stereotypes S representing the Node-RED nodes from TS and declare them in MTS. This step relies on the `require` NodeJS built-in function to load modules;
- Add a function LF to load F in MTS;
- Find the tailored mocked portion from F and declare it in LF;
- Find the control points $CP_1, ..., CP_N$ from F and declare them in LF;
- For each control point CP_i, add a check instruction CI_i in LF;
- Add the instruction to start F at the end of LF.

A simplified test script generated by the aforementioned procedure which corresponds to the test scenario shown in Figure 6 should look like the following:

[12] https://shouldjs.github.io/
[13] http://www.chaijs.com/api/assert/
[14] https://nodejs.org/api/assert.html

```
1 var helper = require("node-red-node-test-helper");
2 var mqttInNode = require("./node_modules/node-red/
      nodes/core/io/10-mqtt.js");
3 var functionNode = require("./node_modules/node-red/
      nodes/core/core/80-function.js");
4 it("Test Scenario 1", function(done) {
5  var flow = [
6  {id: "n0", name: "Mock Reader Device", type: "
      function", func:"msg.payload = {27.5, 26, 24.5};
      flow.set('AVG', 22); return msg;", ..., wires:[["
      n1"]]},
7  {id: "n1", name: "send VALUES to Setter", type: "mqtt
       in", ..., wires:[["n2"]]},
8  {id: "n2", name: "update STATE from VALUES", type: "
      function", ..., wires:[["n3"]]},
9  {id: "n3", name: "check STATE", ..., type: "helper"}
10 ];
11 var nodesTypes = [functionNode, mqttInNode];
12 helper.load(nodesTypes, flow, function () {
13  var mock = helper.getNode("n0");
14  var cp = helper.getNode("n3");
15  cp.on("input", function(msg){
16   cp.context().flow.get("STATE").should.be.equal("
       Warmer Temperature");
17   done();
18  });
19  mock.receive();
20 });
21 });
```

Lines 1-3 are built-in functions referencing the nodes that appear in the flow. The function `it` (line 4) represents a test script in the Mocha environment. Inside the test script, there are the declarations of the flow (lines 5-10) and of the nodes types included in the flow (line 11); some properties have been hidden for space purpose (refer to Figure 4 for a better understanding). Notice the presence of the tailored mocked portion at the beginning of the flow (line 6) and of the control point at the end (line 9); in particular, the control point is of type `helper` from the `node-red-node-test-helper` framework, since for our testing purposes it provides a useful interface for easily recovering information from flows and nodes. Once the flow is loaded (line 12), both the tailored mocked portion and the control point are extracted from it by using their identifiers (lines 13-14), then the control point waits for input events from the function node (line 15) and checks the value of the flow variable `STATE` (line 16), as defined in Figure 6. Finally, the last instruction (line 19) determines the starting of the flow, by means of the tailored mocked portion simulating the reception of a message to be returned to the next node.

We intend to investigate more deeply the procedure for generating Mocha test scripts, in particular in the complex cases which may involve a larger number of flows and different nodes.

4 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a preliminary version of an approach for developing and testing IoT systems in Node-RED. First, the behaviour of the system and the static view are modelled using UML activity and class diagrams, to describe the logics of the system, in terms of nodes and flows, and the main properties of the nodes. Then, from the model is possible to generate the executable Node-RED flows implementing the system and to perform an iterative testing activity aimed at: 1) selecting a set of test scenarios from the model, 2) defining some control points within the selected test scenarios to check over the system properties, and 3) generating the corresponding Javascript test scripts to exercise the selected test scenarios in the Mocha test framework.

In the next future, we intend to: 1) investigate the differences between UML and Node-RED and propose sound guidelines and constraints to support the production of understandable and of high quality models, 2) extend the amount of supported Node-RED nodes, and 3) improve the testing activity with a strategy for selecting effective test scenarios, for generating smart input data tailored for the test scenarios, and for formulating more complex control points. We will implement a tool supporting the tasks of the approach that are intended to be automated (i.e., the generations of the Node-RED flows and the test scripts) and we will evaluate the approach on realistic case studies and compare it against other approaches existing in literature.

REFERENCES

[1] Miroslav Bures, Tomás Cerný, and Bestoun S. Ahmed. 2018. Internet of Things: Current Challenges in the Quality Assurance and Testing Methods. *CoRR* abs/1805.01241 (2018). arXiv:1805.01241 http://arxiv.org/abs/1805.01241

[2] Diego Clerissi, Maurizio Leotta, Gianna Reggio, and Filippo Ricca. 2017. Towards the Generation of End-to-End Web Test Scripts from Requirements Specifications. In *Proceedings of 25th IEEE International Requirements Engineering Conference Workshops (REW 2017)*. IEEE, 343–350. https://doi.org/10.1109/REW.2017.39

[3] Aaron Kane, Thomas Fuhrman, and Philip Koopman. 2014. Monitor based oracles for Cyber-Physical System testing: Practical experience report. In *Proceedings of 44th Annual International Conference on Dependable Systems and Networks (DSN 2014)*. IEEE, 148–155.

[4] Hiun Kim, Abbas Ahmad, Jaeyoung Hwang, Hamza Baqa, Franck Le Gall, Miguel Angel Reina Ortega, and JaeSeung Song. 2018. IoT-TaaS: Towards a prospective IoT testing framework. *IEEE Access* 6 (2018), 15480–15493.

[5] Maurizio Leotta, Davide Ancona, Luca Franceschini, Dario Olianas, Marina Ribaudo, and Filippo Ricca. 2018. Towards a Runtime Verification Approach for Internet of Things Systems. In *Proceedings of 2nd International Workshop on Engineering the Web of Things (EnWoT 2018)*. Springer.

[6] Maurizio Leotta, Diego Clerissi, Dario Olianas, Filippo Ricca, Davide Ancona, Giorgio Delzanno, Luca Franceschini, and Marina Ribaudo. 2018. An Acceptance Testing Approach for Internet of Things Systems. *IET Software* (2018). https://doi.org/10.1049/iet-sen.2017.0344

[7] J Paul Morrison. 2010. *Flow-Based Programming: A new approach to application development*. CreateSpace.

[8] Gianna Reggio. 2018. A UML-based Proposal for IoT System Requirements Specification. In *Proceedings of 10th International Workshop on Modelling in Software Engineering (MiSE 2018)*. ACM, 9–16. https://doi.org/10.1145/3193954.3193956

[9] Gianna Reggio, Maurizio Leotta, Diego Clerissi, and Filippo Ricca. 2017. Service-oriented Domain and Business Process Modelling. In *Proceedings of 32nd ACM/SIGAPP Symposium on Applied Computing (SAC 2017)*. ACM, 751–758. https://doi.org/10.1145/3019612.3019621

[10] Gianna Reggio, Maurizio Leotta, and Filippo Ricca. 2014. Who Knows/Uses What of the UML: A Personal Opinion Survey. In *Proceedings of 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*, Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran (Eds.). LNCS, Vol. 8767. Springer, 149–165. https://doi.org/10.1007/978-3-319-11653-2_10

[11] Gianna Reggio, Maurizio Leotta, Filippo Ricca, and Diego Clerissi. 2015. What Are the Used UML Diagram Constructs? A Document and Tool Analysis Study covering Activity and Use Case Diagrams. In *Model-Driven Engineering and Software Development*, Slimane Hammoudi, Ferreira Luís Pires, Joaquim Filipe, and César Rui das Neves (Eds.). Communications in Computer and Information Science, Vol. 506. Springer, 66–83. https://doi.org/10.1007/978-3-319-25156-1_5

[12] Maria Spichkova, Anna Zamansky, and Eitan Farchi. 2015. Towards a human-centred approach in modelling and testing of cyber-physical systems. In *Proceedings of 21st IEEE International Conference on Parallel and Distributed Systems (ICPADS 2015)*. IEEE, 847–851.