Model-Based Testing for MQTT Applications

Kotaro Tanabe[1], Yoshinori Tanabe[2,3](✉), and Masami Hagiya[1]

[1] University of Tokyo, Tokyo, Japan
[2] Tsurumi University, Yokohama, Japan
`tanabe-y@tsurumi-u.ac.jp`
[3] National Institute of Informatics, Tokyo, Japan

Abstract. Model-based testing is a widely-used vital technique for testing software running in a complex environment. In this paper, we propose extensions to existing model-based tools to apply this technique to software that employs the MQ Telemetry Transport (MQTT) protocol for transmitting messages, commonly used in the Internet of Things (IoT) environment. First, in the finite state machine used for generating test cases in a model-based testing framework, we introduce a type of transition that is triggered when receiving MQTT messages. Second, we extend the finite-state machine so that it produces test cases that reflect the characteristics of IoT software – a large number of relatively simple devices communicate with servers. Third, the concept of time is introduced into the finite state machine. Naturally, this is necessary for verifying the properties of software that runs for a long time. Moreover, to facilitate such verification, both real-time and virtual time are introduced. We implemented these extensions into a model-based testing tool, Modbat, and conducted a small experiment to confirm the feasibility, gaining positive results.

1 Introduction

The model-based testing techniques have now been widely accepted as an efficient testing technique. This technique systematically generates test cases from software models, and hence, high-quality test suites that cover corner cases are obtained. The authors applied this technique to cloud software [1] to show that it fits distributed software.

Along with cloud software, IoT software is also attracting attention in the field of distributed systems. In typical IoT systems, small devices, such as sensors participate. The number of devices is usually large, individual devices are not very reliable, and some are prone to failure. The network connecting these devices and the servers is often unreliable.

The aim of this research is to generate test cases for software that runs in such an environment, such as controlling devices, by sending messages to the devices,

Supported by JSPS KAKENHI Grant Number 17H01719.

receiving messages from the devices, analysing the messages, them or managing the device status. More specifically, we concentrate on software systems that use the MQ Telemetry Transport (MQTT) [2] protocol as our system under test (SUT).

The input to such software is dependent on the status of the environment – devices, network, or other factors that affect the input, such as temperature or time of day. Devices for IoT are usually connected to the Internet wirelessly. However, the network communication may become unstable if the devices are far from the router or base station, or if the devices are mobile. In addition, if the size of the battery or antenna of the devices is small, the risk of malfunction due to battery exhaustion or communication failure increases. Owing to the instability of communication and device operation, a number of devices move asynchronously in IoT systems. These features make it challenging to conduct an integrated test of the entire system.

Model-based testing is potentially a good choice to address the testing of such systems. By modelling these factors with transition systems, we can expect various test cases to be generated. This research proposes extensions for model-based testing to support the testing of IoT systems. Our method consists of three approaches.

First, we support MQTT message communication. We propose a method to describe the subscription at the level of the model and enable setting the arrival of a message as a guard condition.

Second, to model the behaviour of a number of devices, we make it possible to share a model (a finite state machine) for many devices. The number of devices operating in IoT systems tends to be large, and the simulation of the individual behaviour would consume enormous computational resources. Therefore, we propose a method to handle them collectively using a transition system. In our approach, the condition of individual devices is treated as a distribution of states in an integrated model. Through this extension, model-based testing scales to large systems. This extension makes model-based testing scalable to an increase in the number of devices.

Third, we introduce the notion of the timeout, which enables us to describe systems that are affected by time. A timeout is a kind of guard condition that is enabled by the passage of time. Using this extension, the model can describe timing properties, such as waiting for a particular input for a limited time. As an implementation of this extension, it is possible to pass real-time given by a transition system. However, this method causes time to wait without doing anything during the test, which makes the test time-consuming. Thus, we propose an implementation that manages virtual time and skips the time passage. We also propose a function to wait real-time for some part of the test because running the whole test of actual systems in virtual time can cause a problem.

We implemented these extensions in model-based testing tool Modbat [3]. In addition, we created test models using these extensions and conducted experiments to examine our contribution to the speedup of test execution time.

The remainder of this paper is organized as follows. Section 2 provides the background, Sect. 3 describes our proposed extensions to Modbat, Sect. 5 presents the results of experiments, and Sect. 6 concludes.

A previous version [4] of this paper was presented, without peer review, at a Japanese domestic workshop.

2 Background

2.1 Modbat

Our study uses the model-based testing tool Modbat [3,5]. This tool uses a model called the extended finite state machine (EFSM) [6] described in a domain-specific language (DSL) based on Scala [7]. EFSM offers the advantage that transition functions are integrated with the runtime environment, and complex data structures or callback functions can be embedded in the model directly [8].

The application of Modbat to real systems includes testing of libraries with non-blocking I/O and exceptions. A previous study [9] found a bug in the Java network library java.nio, and also a race condition in rupy, a high-performance lightweight HTTP server [10]. Another study [1] detected multiple defects in the Apache ZooKeeper, which is a service for maintaining configuration information, naming, and providing distributed synchronization and group services [11].

2.2 MQTT

MQTT is a widely used transport protocol designed for IoT systems. It is supported as one of the principal protocols by many IoT development platforms such as Amazon AWS IoT [12], Microsoft Azure IoT Hub [13], IBM IoT Platform [14], and many more. MQTT (the latest version as of this manuscript is v5.0) is an official OASIS standard [15].

Compared to traditionally used protocols such as HTML, MQTT is lightweight and thus suitable for devices typically used in the IoT environments because the battery consumption is lower.

An MQTT system consists of a server called a broker and clients. It employs a publish/subscribe model. As an example, let us consider a thermometer as an MQTT client. When it reports the current temperature, instead of sending the data to each client who is interested in it, the thermometer "publishes" the data in a "topic name", such as `temperature/japan/tokyo`. The actual data goes to the broker. Other clients who are interested in the data declare that they "subscribe" to the topic beforehand. Thus, the broker can pass the data to the subscribers.

2.3 Related Work

Model-based testing uses abstract models to generate test cases automatically. Typically, concrete test cases are generated from a test model created by a user. There are many model-based testing tools such as Modbat, Spec Explorer [16] and MaTeLo [17].

In this paper, we used the term "model-based testing" in the sense where test cases are generated from finite state machines. Although many model-based testing tools, such as Modbat, MISTA [18] and MoMuT::UML [19] fall into this category, any technique that is based on models or that involves models are generally called model-based testing. For example, models used in FMBT [20] are written in a language that describes pre-post conditions. Tcases [21] generates test cases from XML documents that describe functions and input value ranges.

Testing MQTT software is a widely researched topic. Many are interested in verifying MQTT brokers and libraries for MQTT clients. In [22], a performance analysis of MQTT is carried out using statistical model checker UPPAAL-SMC.

In [23], a model-based testing technique is applied to MQTT software. The main target is server software, namely, MQTT brokers. A model was obtained through the active automata learning method, then it was used to generate test cases, and they are applied to several MQTT brokers, and found 18 bugs in total. This research is different from ours as our target is not servers but clients of MQTT, but it suggests that model-based testing is suitable for testing MQTT software.

We have previously attempted [24] to incorporate the concept of time into Modbat. In that research, we introduced the modifier `stay` for transitions, meaning that the next transition is disabled for a specified amount of time, after the modified transition. It worked in many cases, but was not compatible with our newly introduced subscription-triggered transition. Therefore, we introduced a new modifier `timeout` with different semantics in this paper.

3 Extensions to Modbat

In this section, we describe our main contributions to this study. We have extended Modbat so that:

– sending and receiving MQTT messages can be written in models.
– the following two important concepts can be written in models:
  • Time
  • Number of devices

In the following subsections, we discuss each of these features.

3.1 Subscription-Triggered Transition

Because our target is software that uses MQTT, our model should be able to handle MQTT messages. Sending messages can be written in methods attached to transitions in the original Modbat system, without any further help. Receiving messages is what we need to handle.

We introduce into the Modbat model a type of transition that is triggered when an MQTT message arrives. In other words, an MQTT message can be regarded as a precondition for invoking this type of transition. An MQTT topic should be specified for a transition. Thus, the transition "subscribes" the topic. Let s be a state of an EFSM and be the source of a transition of this type. If the current state of an EFSM is s and a message of the topic arrives, then

```
class Class1(...) extends Model { ...
  "A" -> "B" := {  ...
    publish("topic1", "msg 1") ...
  }
  "B" -> "B" := { ...
    publish("topic4", "msg 2"); ...
  } subscribe "topic3"
  "B" -> "A" := { ...
  } subscribe "topic2"
  ... }
```

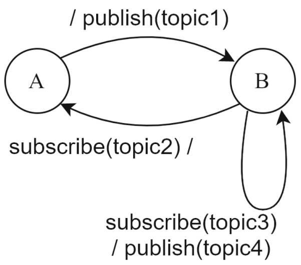

Fig. 1. Subscription-triggerd transition

the transition is invoked, and the message is available in the method attached to the transition. On the other hand, if s is not the current state, the message is ignored (for this EFSM). This behaviour is compatible with the fact that an MQTT client can receive messages only when it subscribes the topic of the messages.

Figure 1 shows a code snippet and a diagram for a model called controller. It has two states A and B. The transition from B to A has a guard `subscribe"topic2"`, meaning that it is only fired when an MQTT message with topic `topic2` is received. The transition from A to B calls `publish` in its attached method, which sends a message to a running MQTT broker.

3.2 State Distribution

In frameworks of model-based testing based on extended finite system machines (EFSM), test cases correspond to paths in the EFSM. To generate test cases on the fly, Modbat keeps the "current state" and moves it along enabled transitions. We can have two or more EFSMs for a system when we have several objects to be considered to generate test cases. In such cases, each EFSM has its current state.

In our application of Modbat for testing MQTT-based software, it is natural to model the behaviour of an IoT device (MQTT clients) with an EFSM. As mentioned above, it is already possible for the original Modbat to have an EFSM for each MQTT client. However, there are several issues to be considered.

First, in IoT environments, the number of MQTT clients may increase. They share an EFSM, but the current states are different. We may be interested in the situation as a whole rather than the state of individual devices. For example, if the EFSM has two states representing normal and failure, we may be interested in the number of devices that are in the failure state. The current Modbat EFSM is not suitable as it assumes independent EFSMs.

Second, in the current implementation of Modbat, a dedicated thread is allocated to each EFSM. Therefore, simulating many devices will result in poor performance.

```
"A" -> "X" := { ... } subscribe(topic1)
"A" -> "Y" := { ... } subscribe(topic2)
"A" -> "Z" := { ... } timeout(10*Const.min)

"B" -> "X" := { ... } timeout(2*Const.hour, 8*Const.hour)

"C" -> "X" := { ... } realTimeout(30*Const.sec)
```

Fig. 2. Transitions with timeout

To address these issues, we have introduced a new type of EFSM. Conceptually, it is a collection of EFSMs that share the same states, transitions and actions. Therefore, it has many current states, or in other words, each state keeps the number of "instances" that stay on the state. The number of instances in each state can be retrieved and used to describe specifications; such as "the ratio of broken devices is less than 3%". Moreover, in order to handle the timeout behaviour (see Sect. 3.3 for details), Modbat tracks the number of instances that enter the state in each time slice.

3.3 Timeout

Many systems behave depending on time. Some sensor systems may send their reports periodically, for example, once in an hour. Some control systems may wait for a message from devices that it monitors for a certain period of time, and if no message comes in, it may judge that the device is out of order.

To describe these types of system behaviour, we introduced the concept of timeout into the Modbat model.

Formally, timeout is a property of transitions, with the amount of time. For example, in Fig. 2, a ten-minute timeou t is attached to the transition from A to Z. The transition is fired 10 min after an instance enters A if no message with topic *topic1* or *topic2* is arrived during the period. Another type of timeout has two parameters, such as the one from `B` to `X`. In this case, the expiration time is chosen randomly between 2 h and 8 h.

One state can have at most one transition with timeout attached, so there are no races between transitions.

3.4 Virtual and Real Timeout

Waiting for all timeouts in real time would not be realistic. We cannot wait for four hours when we conduct a system test. An obvious workaround would be to reduce time by a fixed percentage, but it is not ideal for the following reasons.

First, even with a reduced rate, we still suffer from unnecessary waiting time, which will not become zero. Second, some timeouts absolutely require the specified amount of time. For example, we have MQTT message exchanges between our running models, and it takes some time for a message to be delivered.

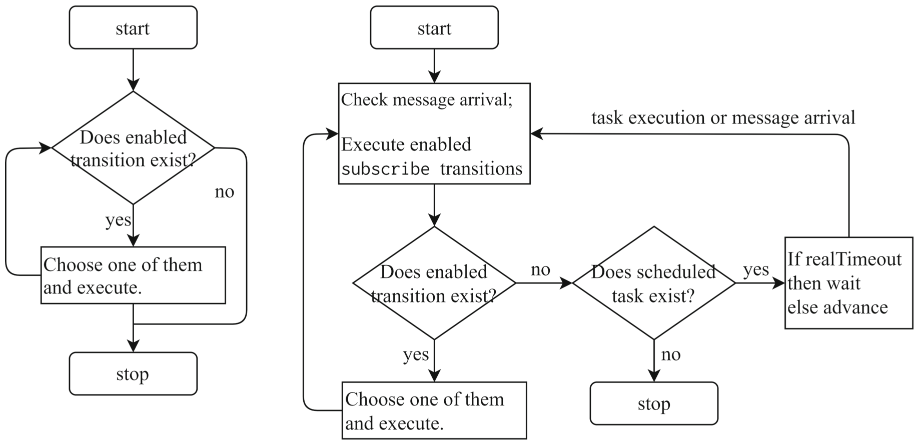
start
Does enabled transition exist?
yes
no
Choose one of them and execute.
stop
start
Check message arrival;
Execute enabled subscribe transitions
task execution or message arrival
Does enabled transition exist?
no
Does scheduled task exist?
yes
If realTimeout then wait else advance
yes
Choose one of them and execute.
no
stop

Fig. 3. Changes in the main loop of Modbat

Therefore, if we try to decide whether a device is in failure state by sending a message and receiving a response, we need to wait for a specific time, and it cannot be reduced for speed-up.

Therefore, we have introduced two different types of timeout: real and virtual. A real timeout expires only when the specified amount of time elapses physically. On the other hand, when a virtual timeout is stored, we record the expiration time for the timeout. When the system enters a status where no other transition can be (immediately) invoked, then all the pending virtual timeouts are checked, and the current time is "skipped" to the earliest expiration time.

We also have EFSMs with more than one instance, as described in Sect. 3.2. Instances that share the same current state may have different expiration times. Therefore, simply counting the number of instances in a state is not sufficient. However, if we kept expiration time for each instance independently, the advantage of collecting instances would be lost. To address this issue, we internally define a value *timeSlice* and split time with this interval. Instances that fall into the same interval are regarded as a group, and a representative timeout expiration time is recorded for the group.

4 Implementation

The introduction of three extensions had a major impact on the main loop of Modbat. The original main loop, shown in the left-hand side of Fig. 3, is rather simple: it chooses one of the enabled transitions randomly and executes it, while there are such transitions.

We modified the main loop is as follows: It first chooses a state with enabled transitions (without timeout or subscription), if there is some. If two or more enabled transitions exist, it divides instances staying on the state according to the weight (probability) of the transitions, and execute them. Once, it becomes a situation where there are no enabled transitions, Modbat checks whether scheduled tasks (transitions with timeout whose source state has waiting instances) exist. If all scheduled tasks are in virtual time, Modbat advances the virtual time to the earliest expiration time. If there is one with a real timeout, Modbat sleeps until the earliest timeout expiration time (including virtual time). If a message arrives while sleeping, Modbat wakes up and fires transition that subscribes the topic of the message. The right-hand side of Fig. 3 illustrates the modified main loop.

A task is scheduled when a model instance enters a state, which does not have enabled transitions but has a transition with a timeout. If the timeout has only one expiration time, it is simply recorded with the number of model instances. If the timeout has two parameters, the model instances are evenly divided into groups according to the value of *timeSlice*, which is configurable by the user, and the expiration time for each group is recorded.

The virtual timeout is implemented using an Akka mock scheduler [25]. This scheduler provides basic features for handling virtual time, such as skipping a fixed amount of time. We implemented the necessary features for Modbat, such as advancing the virtual time spontaneously, on top of this scheduler.

5 Evaluation

To demonstrate the effectiveness of our methods, we conducted a small experiment. First, we designed a small MQTT application and built the corresponding Modbat models to confirm that the features introduced in this paper are sufficient to describe the testing environment. Second, we ran the model to see the following:

– whether the virtual timeout works as expected so that the runtime is shorter than the case of real time,
– whether the multi-instance EFSM works as expected so that it reduces the runtime compared to many single-instance EFSMs.

These experiments test a system of smart wattmeters. This system consists of meters and a controller communicating through MQTT. The meters measure electric power in all rooms in a house. The controller receives the data, sums them, and reports the total electrical power to the user. There is a probability that some meters may be broken. When the controller detects some broken meters, it sends an alarm to the operator of the system.

In these experiments, we tested the behaviour of the controller. We verified that the controller reports the correct value and that it provides a warning if and only if some meters are broken.

5.1 Models

A model of the smart house consists of five types of model instances, as shown in Figs. 1, 2 and 4. Meter has multiple instances, while the other four models have only one instance each.

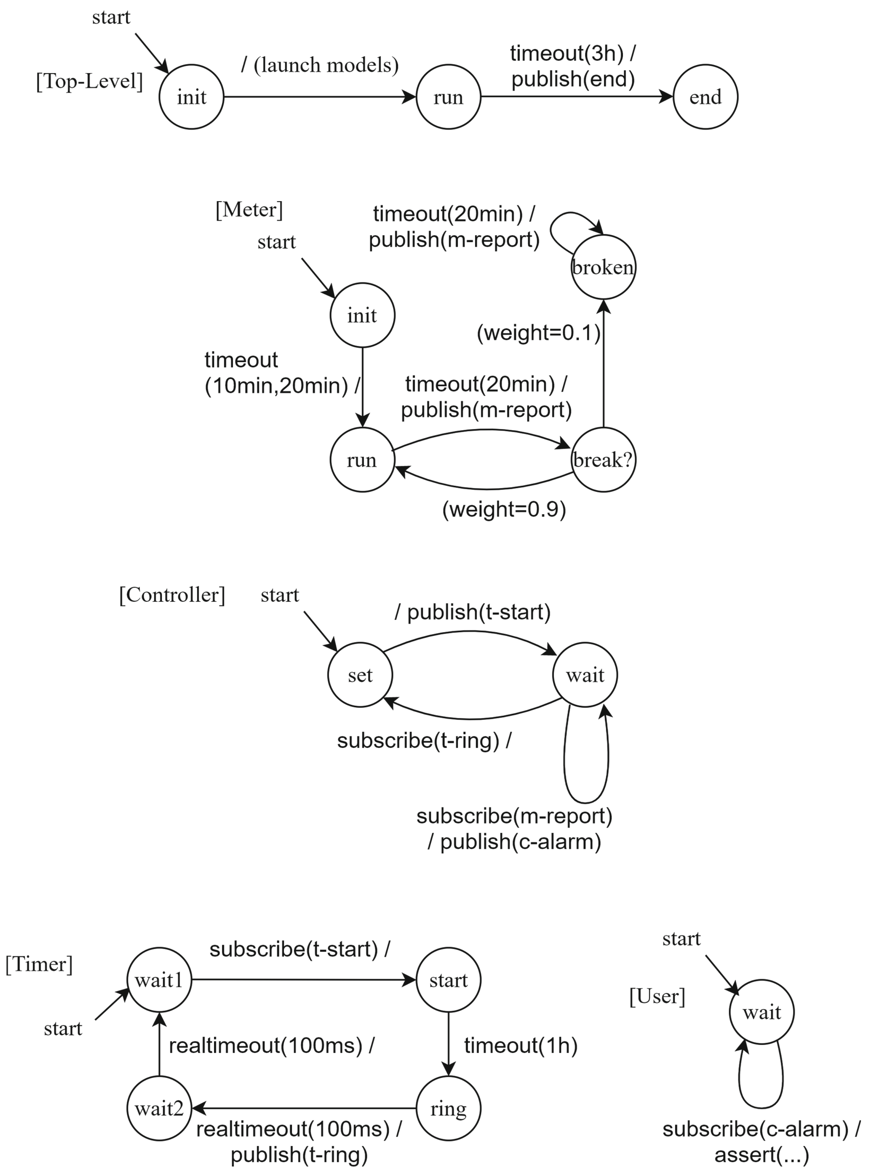

Fig. 4. Models used in the experiments

Top-Level. The instance of this model is created at the start of the simulation. It launches instances of other models in the transition from state *init* to state *run*. After waiting for some virtual time in *run*, it sends an MQTT message that notifies the end of the test session to a topic called `end`. All the states in other models have a transition subscribing to this topic, which leads an instance to the final state of the model. (Related transitions and states are omitted in the figures.)

Meter. A meter is installed in each room. At first, all meters are running correctly. Each meter measures the electric power in the room, and reports it to the controller every twenty minutes, using topic `m-report`. After the transition, each meter returns back to the *run* state with probability 90% and goes to the *broken* state with probability 10%, which means the meter is broken. Meter instances in the *broken* state report watt to the controller repeatedly like correctly-running meters, but they report an incorrect value. If a meter is broken, it will not be repaired during the simulation.

Controller. The controller receives messages from meters, sum them up, and reports the result to the user every hour. First, the controller sends a message to the timer using topic `t-start`. Then, it waits until the timer notifies the passage of one hour using a message with topic `t-ring`. While waiting, the controller receives values from the meters through messages with topic `m-report` and sums them up. In addition, if the received value is much higher or much lower than the average of previously received values, the controller assumes that the meter that sent the irregular value is broken, and publishes an alarm message to the user using topic `c-alarm`. After a message from the timer arrives, the controller reports the summed value to the user.

Timer. The timer starts when it receives a message with topic `t-start`. After waiting for one hour by virtual time, it publishes a message that tells the passage of one hour with topic `t-ring`. Considering the time of MQTT communications, the timer waits for a brief duration of real-time before and after publishing the message.

User. This is a model of a tester of the controller. The user subscribes topics `m-report` and `c-alarm`. It asserts that the value reported to `m-report` is correct and that the controller publishes a message to `c-alarm` if and only if some meters are broken.

5.2 Experiments and Results

We conducted two experiments to demonstrate the effectiveness of the methods described in the previous chapter. The experiments show the speedup of the test execution time by introducing virtual time and the shared model. We performed experiments on Ubuntu 16.04 on Oracle Virtual Box. The host OS is Windows 10 Pro, running on a machine with Intel Core i7-6600U CPU and 8GB memory.

To evaluate the effect of virtual time on test time, we run the tests of the models described above with two variants. One uses keyword `timeout` for the timeout of the transition from state `run` to `end` in model *Top-Level* (from now on referred to as transition `end-trans`), which means the test is executed in virtual time. The other uses keyword `realTimeout` for transition `end-trans`, which means the test is executed in real time. We have converted the value of timeout of `end-trans` and have run tests for each value.

The results are shown on the left-hand side of Table 1. Execution time is the average of five runs. The execution time of the code with `timeout` is shortened, while that of the code with `realTimeout` is approximately the same as the timeout of `end-trans`.

Table 1. The results of experiments

Amount	Execution time (sec)	
	Virtual	Real
1 min	3.9	62.2
10 min	3.5	602.2

# of devices	Execution time (sec)	
	Indiv.	Shared
10	3.1	3.3
100	20.0	13.3
200	37.0	18.4
1000	137.4	36.4

We also conducted an experiment to evaluate the effect of introducing models with two or more instances. In this experiment, we run the tests of the same models as the first experiment with two types of source code. The two codes are different in how to handle the model of devices. One operates the instances of the meters individually, and the other operates a distribution of the meters.

The results are shown on the right-hand side of Table 1. As the number of devices increases, the execution time is faster when managing by distribution in comparison to running instances individually.

6 Conclusion

We proposed extensions for model-based testing for IoT systems and implemented them in a model-based testing tool Modbat. The first extension makes it possible for transitions to be fired when an MQTT message is received. By the second extension, we can have models that reflect the status of many instances. Two types of timeout features, real and virtual, are introduced as the third extension. We have built a small experimental test environment that uses the extensions and confirms that it works as expected functionally and shows performance improvements compared to the original version.

Future tasks include providing a virtual time library for SUTs. Currently, if an SUT issues the sleep method, then it naturally sleeps real-time. Instead, if the SUT uses the library, virtual time is applied when running under Modbat while keeping the original behaviour (sleep) on production.

Another direction is implementing dedicating MQTT broker for Modbat. This will allow the user, when testing the MQTT broker is out of the scope, to remove real-time timeout for communication, and enable Modbat to run completely in virtual time. In addition, this extension will make it easier to simulate an unstable network by controlling packet loss or delay of delivery in Modbat.

Acknowledgements. The authors would like to thank Professor Cyrille Artho for many helpful discussions and valuable comments.

References

1. Artho, C., Gros, Q., Rousset, G., Banzai, K., Ma, L., Kitamura, T., Hagiya, M., Tanabe, Y., Yamamoto, M.: Model-based API testing of Apache ZooKeeper. In: International Conference on Software Testing, Verification and Validation (ICST 2017), pp. 288–298 (2017)
2. Rahul, G., Andrew, B.: MQTT version 3.1.1. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html
3. Artho, C., Biere, A., Hagiya, M., Platon, E., Seidl, M., Tanabe, Y., Yamamoto, M.: Modbat: a model-based API tester for event-driven systems. In: Hardware and Software: Verification and Testing, pp. 112–128 (2013)
4. Tanabe, K., Tanabe, Y., Hagiya, M.: Speedup of model-based testing for IoT software using virtual time and state distribution of devices. IEICE Tech. Rep. **119**(392), 37–42 (2020)
5. Artho, C., Biere, A.: Modbat. https://people.kth.se/~artho/modbat/
6. Cheng, K.T., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: Design Automation Conference, DAC 1993, pp. 86–91 (1993)
7. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima, Mountain View (2008)
8. Artho, C., Havelund, K., Kumar, R., Yamagata, Y.: Domain-specific languages with Scala. In: Formal Methods and Software Engineering, pp. 1–16 (2015)
9. Artho, C., Hagiya, M., Potter, R., Tanabe, Y., Weitl, F., Yamamoto, M.: Software model checking for distributed systems with selector-based, non-blocking communication. In: International Conference on Automated Software Engineering (ASE), pp. 169–179 (2013)
10. Larue, M., Martino, E., Funk, M., Chen, A., Lee, A., Lung, C., Hoyt, D., Baghdasaryan, H.: rupy - a tiny Java nio HTTP application server (2013)
11. Hunt, P., Konar, M., Grid, Y., Junqueira, F., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: ATC. USENIX, vol. 8 (2010). Yahoo Research
12. Barr, J.: AWS IoT – cloud services for connected devices. https://aws.amazon.com/blogs/aws/aws-iot-cloud-services-for-connected-devices/
13. Communicate with your iot hub using the MQTT protocol. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-mqtt-support
14. Getting to know MQTT. https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/
15. MQTT v5.0 OASIS standard. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html
16. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, pp. 39–76 (2008)

17. Model based testing tool - discover MaTeLo — ALL4TEC. https://www.all4tec.com/
18. Xu, D., Thomas, L., Kent, M., Mouelhi, T., Traon, Y.L.: A model-based approach to automated testing of access control policies. In: Proceedings of ACM symposium on Access Control Models and Technologies (SACMAT 2012), pp. 209–218 (2012)
19. Krenn, W., Schlick, R., Tiran, S., Aichernig, B., Jobstl, E., Brandl, H.: MoMut::UML model-based mutation testing for UML. In: Proceedings of International Conference on Software Testing, Verification and Validation (ICST), pp. 1–8 (2015)
20. FMBT. https://01.org/fmbt/
21. Tcases. https://github.com/Cornutum/tcases
22. Houimli, M., Kahloul, L., Benaoun, S.: Formal specification, verification and evaluation of the MQTT protocol in the Internet of Things. In: International Conference on Mathematics and Information Technology (ICMIT), pp. 214–221 (2017)
23. Tappler, M., Aichernig, B.K., Bloem, R.: Model-based testing IoT communication via active automata learning. In: Proceedings of International Conference on Software Testing, Verification and Validation (ICST), pp. 276–287 (2017)
24. Yoneyama, J.: Model-based testing simulating unstable networks and devices for IoT software. Master's thesis, University of Tokyo (2018)
25. Github - miguno/akka-mock-scheduler: a mock akka scheduler to simplify testing scheduler-dependent code. https://github.com/miguno/akka-mock-scheduler