Model-based Network Fault Injection for IoT Protocols

Jun Yoneyama[1], Cyrille Artho[2], Yoshinori Tanabe[3] and Masami Hagiya[1]
[1]*Dept. of Computer Science, The University of Tokyo, Tokyo, Japan*
[2]*School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, Stockholm, Sweden*
[3]*Dept. of Library, Archival and Information Studies, Tsurumi University, Yokohama, Japan*

Keywords: Software Testing, Model-based Testing, Model-based Simulation, Fault Injection, Internet of Things, MQTT.

Abstract: IoT devices operate in environments where networks may be unstable. They rely on transport protocols to deliver data with given quality-of-service settings. To test an implementation of the popular MQTT protocol thoroughly, we extend the model-based test framework "Modbat" to simulate unstable networks by taking into account delays and transmission failures. Our proxy-based technology requires no changes to the IoT software, while the model allows the user to define stateless or stateful types or fault patterns. We evaluate our methods on a client-server library for MQTT, a transport protocol designed for IoT.

1 INTRODUCTION

In this paper, we extend methods of model-based testing for testing Internet of Things (IoT) protocols in an unreliable network environment. Model-based testing is suitable to generate many different test scenarios, and may cover many execution paths. However, while it is easy to describe the behavior of each device in model-based testing, it is difficult to describe the behavior of the whole system.

To this end, we use models to control a simulation environment that injects network faults. Our work is integrated in the open-source model-based testing tool Modbat (Artho et al., 2013) and combines two approaches:

1. We enhance Modbat with three extensions, which add a notion of time, facilitate modeling variable error rates, and simplify error handling.
2. We simulate unstable networks during test execution. Our mechanism simulates instability using a software layer that enables easy fault injection.

Compared to existing work, we apply the simulation directly to implementations of IoT protocols, and inject faults entirely in a software environment, without modifying the system under test. Our approach thus allows us to evaluate the robustness of the MQTT implementation.

This paper is structured as follows. Section 2 describes the background of this research and related work. Section 3 describes the definitions and implementations of our extensions. The methods and the results of the experiments are described in Section 4. Section 5 concludes and discusses future work.

2 BACKGROUND AND RELATED WORK

This section gives background and describes related work on model-based testing and simulation, fault injection, IoT, and the MQTT protocol.

2.1 Model-based Testing

Software testing is a method to find defects in software by executing parts of the software. The target software for the test is called system under test (SUT). A test case consists of an input and an expected output, which is compared with the actual output.

Model-based testing uses abstract models to generate test cases automatically (Utting et al., 2012). Typically, many concrete test cases can be derived from a given test model. There exist many tools for model-based testing, such as QuickCheck (Claessen and Hughes, 2000), ScalaCheck (Nilsson, 2015), SpecExplorer (Veanes et al., 2008), and Modbat (Artho et al., 2013).

Yoneyama, J., Artho, C., Tanabe, Y. and Hagiya, M.
Model-based Network Fault Injection for IoT Protocols.
DOI: 10.5220/0007618102010209
In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, pages 201-209
ISBN: 978-989-758-375-9

2.2 Model-based Simulation

Model-based simulation derives a simulation from an abstract model, typically using graph transformation techniques (Kosiuczenko and Lajios, 2007; Torrini et al., 2010). This technique has been applied to simulate networks (Khan et al., 2009) or mobile systems (Heckel and Torrini, 2010), among many others. The differences to model-based testing are that the stimuli to the SUT are indirect, via a simulated environment rather than a direct input, and that the exact output is usually not known.

Models have also been used to directly analyze the correctness or performance of IoT protocols, for example MQTT (Houimli et al., 2017) and related protocols (Alvi et al., 2015; De Rubertis et al., 2013). Our work differs in that we analyze the performance of a running system rather than an abstract model.

2.3 Fault Injection

Fault injection introduces errors into the system artificially in order to test the robustness of the system (Ziade et al., 2004), (Hsueh et al., 1997). Conceptually, fault injection is related to mutation testing (Jia and Harman, 2011) or fuzz testing (Oehlert, 2005) in that mutations or changes to inputs are applied at the protocol layer, to simulate faults in the behavior of the network or some of the nodes sending data on it (Dadeau et al., 2011; Jing et al., 2008; Zhang et al., 2012).

Closely related to our work is an evaluation of the effect of network errors on a Myrinet network interface hardware by using simulated and software-implemented fault injection (Stott et al., 1998). Both methods modify the system under test (SUT) to inject faults. Our work differs in that we inject faults in an intermediary component, without modifying the SUT. This is similar to jepsen (Kingsbury, 2018), which injects faults at the level of system virtualization by parametrizing various aspects of system reliability for the duration of testing.

2.4 IoT and Testing

IoT devices are expected to operate in a potentially unstable network environment; hence, much work exists on testing IoT-related transport protocols and their impact on systems. Transport protocols designed for IoT devices include MQTT (Banks and Gupta, 2014) and CoAP (Bormann, 2016).

MQTT uses *publish/subscribe* protocol in which multiple clients exchange messages via a server, called "broker". Each message has a payload, a *topic name* and a value named *Quality of Service* (QoS). Our work focuses on QoS. MQTT protocol can guarantee message arrival even if devices are working in an unstable network environment where TCP connections may be lost.

MQTT defines three types of QoS, 0 to 2, as follows:

QoS 0. Each message is delivered at most once. The message may be lost during delivery.

QoS 1. Each message is delivered at least once. The message may be delivered multiple times.

QoS 2. Each message is delivered exactly once. No message loss or redundant delivery is allowed. (Banks and Gupta, 2014)

Past work has described the MQTT protocol formally (Mladenov et al., 2017) in a test description language, TTCN-3 (Grabowski et al., 2003). In this study, and MQTT servers are tested according to a set of fixed test cases that are derived the specification. Existing work tested three MQTT server implementations and found that all of them produced responses violating the protocol specification. Other protocols and features are covered by ongoing work (Testware, 2018). In contrast to this work, our models generate variable test outcomes.

Model-based testing has also been applied to MQTT (Tappler et al., 2017) and other protocols (Hsu et al., 2008), with the result of uncovering inconsistencies between various implementations. Other studies evaluate the correlation between message size, network instability, and performance, by varying network delay and packet loss rate (Thangavel et al., 2014; Lee et al., 2013). Our approach differs in that no extra equipment is required, as the network environment is entirely controlled by software.

2.5 Extended Finite State Machines

An extended finite state machine (Cheng and Krishnakumar, 1993) is defined as a 7-tuple $M = (I, O, S, D, F, U, T)$, where S is a set of states, I is a set of input symbols, O is a set of output symbols, D is an n-dimensional vector space $D_1 \times \ldots \times D_n$, F is a set of *enabling functions* f_i s. t. $f_i \colon D \to \{0,1\}$, U is a set of *update functions* u_i s. t. $u_i \colon D \to D$, and T is a transition relation s. t. $T \subseteq (S \times F \times I) \times (S \times U \times O)$ (Cheng and Krishnakumar, 1993). Past work describes details on EFSMs as used by the model-based tester Modbat (Artho et al., 2013), which we use here.

We propose several extensions to EFSMs for modeling an IoT simulation environment. The time extension of EFSMs results in a notation that

is conceptually similar to probabilistic timed automata (Beauquier, 2003). The differences are that with our notation, the time interval for a transition can be a random value between two boundaries rather than a fixed amount, and that our model allows probabilities to change at run time. Furthermore, our modeling notation is optimized for testing network software (Artho et al., 2013; Artho et al., 2017), rather than model checking algorithms (Bulychev et al., 2012) or protocols (Kwiatkowska et al., 2002; Kwiatkowska et al., 2011).

Compared to similar probabilistic extensions of EFSMs proposed in the past (Park and Miller, 1997; Paz, 1971), our notion includes changes of transition probabilities at run-time and has a different notion of parallelism, by interleaving multiple state machines instead of having "fork" states in a single state machine. Our notation of parallelism is inspired by the one used in parallel statecharts (Mikk et al., 1997), which has influenced the current generation of model-based testing tools (Veanes et al., 2008).

2.6 Modbat

We use Modbat (Artho et al., 2013; Artho and Biere, 2018) in our work. Modbat models use extended finite state machines that are described in embedded domain-specific language (DSL) based on Scala (Odersky et al., 2008). This has the advantage that the transition functions are seamlessly integrated with the run-time environment, and that complex data structures and callback functions can be directly embedded in the model (Artho et al., 2015; Artho et al., 2013). Furthermore, due to Modbat being based on Scala, it is very easy to extend the platform with new features or annotations, because embedded DSLs use the parser of the host language, Scala.

Test generation in Modbat is done online, and starts with the initial state of the initial *model instance*, which implements an EFSM. From the current state of that model instance, a transition is chosen at random from all enabled outgoing transitions. Transitions are enabled or disabled by their preconditions. Whenever a transition is executed, the transition function may call the SUT, and also check the result of the call. A transition may carry an optional *label*, which allows the model to refer a transition by name. Finally, transitions may create and *launch* new model instances, which start at their own initial state. If multiple model instances are active, one transition from all models is chosen at random at each test step. Modbat therefore implements an interleaving semantics when deriving tests from concurrently active models (Artho et al., 2013; Artho et al., 2017).

In this work, we use Modbat both for model-based testing (to execute MQTT implementations), and for model based-simulation, to control fault injection in the network simulation environment.

3 METHODS

This section describes our extensions for model-based testing and simulation in Modbat. Section 3.1 gives an informal description of our extensions and their implementation, and Section 3.2 shows an example. Section 3.3 describes a new way of injection faults into the network environment.

3.1 Modbat Model Extensions

We introduce extensions to model transitions dependent on time intervals, transitions whose probabilities may change over time, and another extension to handle callbacks more flexibly.

3.1.1 Time

Standard EFSMs (Cheng and Krishnakumar, 1993) are not affected by time. To model temporary faults in a system, we want to be able to temporarily suspend a transition, for a random amount of time within a bounded interval.

We implement this feature with a new annotation `stay(x,y)`. The annotation applies to a given transition t and affects all future transitions of that model instance. The annotation takes two arguments, which specify the time interval within which the model instance is suspended after t has been executed. If all active model instances are suspended for a certain amount of time, test execution resumes after the first timer has expired.

This extension facilitates modeling temporary faults but also covers the case where all model instances are temporarily suspended. In the original version of Modbat (Artho et al., 2013), preconditions can disable a transition temporarily. However, if no precondition is satisfied, test execution terminates at that point. This was because preconditions were assumed to be state-based and not time-based (Artho et al., 2013). Our new annotation prevents the problem of terminating tests prematurely in such cases.

3.1.2 Dynamic Weight Change

An EFSM in Modbat behaves probabilistically in actual test executions according to the *weights* of the transitions. The probability of a transition being chosen is proportional to its weight. Fixed transition

weights are useful in generating the desired distribution of test cases, but only variable transition weights can reflect temporary changes in the test setting.

In Modbat, weights are expressed by annotation `weight`, which takes a `double` as an argument. Our extension updates the weight of all functions with matching labels. It is implemented by a method `setWeight(l,w)`, which takes a string l and a double value w, and sets the weights of the transitions whose labels correspond to l, to w.

In the original version of Modbat, this feature would have to be expressed using extra preconditions that use random choices to simulate the same effect. Our extension allows the simulation of variable error rates in a very clear way.

3.1.3 Transition Invocation

Extended finite-state machines can describe inputs and expected (synchronous) outputs well, but callback functions have to be modeled outside the transition functions themselves (Artho et al., 2017). When describing side effects of a callback, extra variables have to be used to model its effect on the model state, and some actions may even affect the global program state (i. e., the state of another model). In order to model callback functions in a more straightforward way that does not require auxiliary variables to record the mere occurrence of a callback, we introduce an extension that invokes transition functions directly.

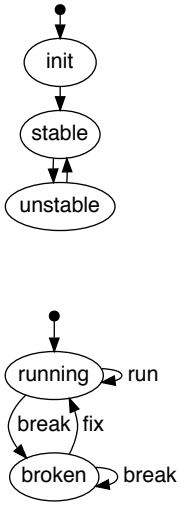
In this extension, we identify transitions by their labels. As callbacks are typically asynchronous, their effect on the EFSM is not immediate, but registered in a queue q of pending transitions. A method `invokeTransition(l)` adds a label l to q. When called from inside a callback function, that method is typically executed in a separate (callback) thread, while test case generation is controlled by Modbat in the main thread (Artho et al., 2017). Pending transitions in q have priority over regular transitions, so the effect of a callback is processed as soon as the currently executing transition has finished. This prioritization would be difficult to implement with multiple model instances without our extension in Modbat. Capturing this feature in the model itself would require an extra precondition in each transition, to check whether any transition invocation is pending.

3.2 Example

Figure 1 shows an example of models of the environment and a device. Here, `skip` is defined in Modbat as a method doing nothing. At the time the device is launched, it goes to state "running". There is a small chance of the device breaking, indicated by weight 0.1

```
class Environment extends Model {
  var device: Device = _
  "init" -> "stable" := {
    device = new Device
    launch(device)
  }
  "stable" -> "unstable" := {
    device.setWeight("break", 1.0)
  }
  "unstable" -> "stable" := {
    device.setWeight("break", 0.1)
  }
}
class Device extends Model {
  "running" -> "running" :=
    skip label "run"
  "running" -> "broken" := {
    // some actions
  } label "break" weight 0.1 stay 5000
  "broken" -> "running" :=
    skip label "fix" stay 0
  def callbackOnReset(): Unit = {
    invokeTransition("fix")
  }
}
```

(a) Source code



(b) Diagrams

Figure 1: Example Modbat models of the environment and a device.

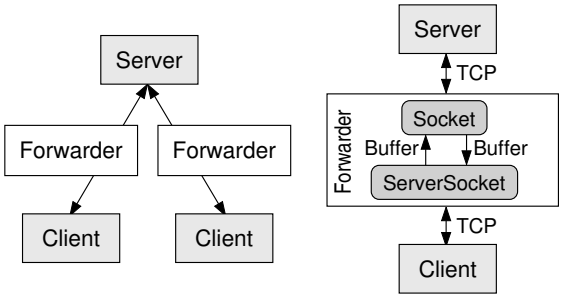
in that transition (the default weight is 1.0). When the model of the environment goes to state "unstable", the weights of the transitions in the device model are updated, and the model stays in state "running" or goes to state "broken" with equal probability.

Once the device is broken, it stays broken for 5000 milliseconds. In addition to recovering after some delay, the device may also be reset, at which point callback function `callbackOnReset` is called. At this point, the label "fix" is pushed to the queue. When that transition is executed, it resets the state to "running" and clears the pending timer.

3.3 Packet Forwarder

In order to test a program under unstable network environments, we propose a *fault injection* (Hsueh et al., 1997) method to simulate network errors and delays. That method is applicable to software in which servers and clients communicate one another via TCP/IP, and requires no modification of the software under test.

We simulate an unstable network by inserting a mechanism named *packet forwarder* between a server and a client as shown in Figure 2a. The packet forwarder forwards packets to both directions, from the server to the client and the client from the server. Network errors and delays can be injected by controlling the forwarder. The order of the packets is not modi-



(a) Usage of packet forwarders

(b) Internals of a forwarder

Figure 2: Packet forwarders.

fied in this method, because packet ordering is guaranteed by TCP. Because the SUT is not modified, our method preserves its original behavior: Each injected network fault could also happen in a real environment.

Figure 2b shows the implementation of our packet forwarder. `Socket` and `ServerSocket` are instances of classes `java.net.Socket` and `java.net.ServerSocket`, respectively. The `Socket` acts like the client to the server, and the `ServerSocket` acts like the server to the client. The forwarder has two buffers, corresponding to bidirectional communication. Each buffer stores the contents of the packet from one socket and sends them to the other socket.

When a connection loss is simulated, we close each TCP connection and port by calling close on `Socket` and `ServerSocket`. When a network delay is simulated, the threads forwarding data between the sockets sleep for a particular time duration. Therefore, network delays in each direction can be controlled independently.

We associate a model instance in Modbat with each packet forwarder; the configuration of the packet forwarder is controlled by the model. Thus, in addition to providing inputs to the SUT, the models also manage the state (configuration) of the simulation. They switch between modes where nominal behavior is tested, and modes where faulty components and network problems are simulated. A model can also observe the state or output of the SUT, and abort a test if a property violation is detected.

4 EVALUATION

In order to show the validity and effectiveness of the methods described in Section 3, we test MQTT client/server combinations under unstable network environments. In particular, we evaluate the following question: Can the packet forwarder uncover the effects of missed packets in the SUT?

Experiments were done on a machine running Ubuntu 16.04 with Intel Core i7-3770 and 16 GB memory. We use Java 1.8.0_161, Scala 2.11.8, Mosquitto 1.4.14 (Light, 2017), and Eclipse Paho 1.2.0 (Eclipse Paho Team, 2018), and an extended version Modbat 3.2 as shown in Section 3.

4.1 Method

In each experiment in this section, the modeled system has an MQTT server and two MQTT clients. The two clients are named "sender" and "receiver" respectively, and both of them are connected to the server via packet forwarders.

A test case is executed in the following way.

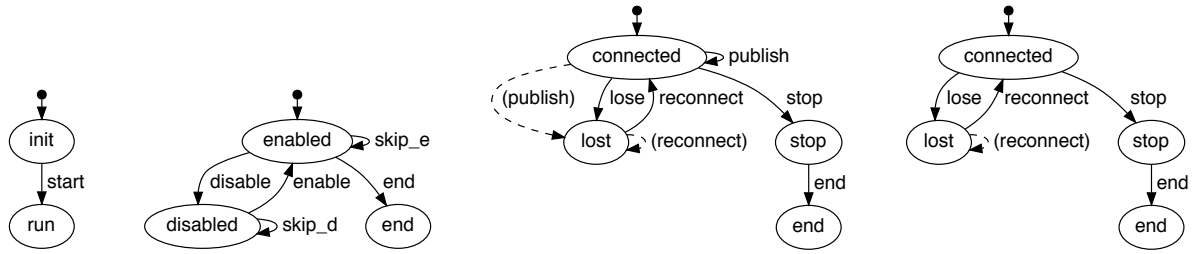
1. The server is started.
2. The sender and the receiver connect to the server.
3. The receiver subscribes to particular topics.
4. The sender publishes several messages to the server; the server forwards them to the receiver.
5. The sender and the receiver shut down.
6. The number of the published and received messages are compared.

To simulate unstable network environments in our MQTT tests, packet forwarders are inserted between each client and the server, so that all network connections are virtualized. We conducted nine experiments, combining three types of QoS and three types of settings for the packet forwarders. In each experiment, messages are published with QoS 0, 1 or 2. The packet forwarder of the stable client is always alive, while the forwarder of the unstable client sometimes cuts the connection between the client and the server.

In these experiments, each of the sender and the receiver runs an instance of class `MqttAsyncClient` in Paho. The Mosquitto server executes outside the test tool. Because we cannot easily determine which of the server and the clients has bugs from the result, we consider both Mosquitto and Paho as SUTs for these experiments.

4.2 Models

Main Model. A model for this experiment consists of four types of model instances. The main model (see Figure 3a) creates model instances of sender and receiver, and launches them.



(a) Main (b) Forwarder (c) Sender (d) Receiver

Figure 3: EFSMs in the model for testing MQTT implementations.

Forwarder. The packet forwarder model (see Figure 3b) has two main states, `enabled` and `disabled`. These two states simulate network environments of variable reliability. The two `skip` transitions have no effect, while transition `disable` stops forwarding packets and closes the connections between the server and the forwarder, and the client and the forwarder. Executing transition `enable` reconnects the forwarder with the server, and the client and forwards data again. The `end` transition terminates operations. Its weight is set to 0.0; therefore, this transition is never executed unless it is explicitly invoked by `invokeTransition`.

Sender. The sender model (see Figure 3c) creates a forwarder instance and launches it. After that, sender connects to the server via the forwarder. As shown in the figure, the sender repeats publishing messages when in state `connected`. Method `publish` in `MqttAsyncClient` returns a *token*, which stores the progress of the message delivery. The model stores all the tokens in a list. If the sender tries to publish a message while the forwarder is disabled, `publish` in Paho throws an exception, and the model goes to state `lost`. (The dashed arrow indicates a non-deterministic outcome of an operation that usually but not always succeeds.) Transition `lose` with weight 0.0 records a connection loss and is invoked by a callback function in `MqttAsyncClient`. When the model is in state `lost`, the sender tries to reconnect to the server. If the reconnection succeeds, the receiver is `connected` again, otherwise it stays in state `lost`.

The model also randomly goes from state `connected` to state `stop`. In this transition, the sender prepares to disconnect. Concretely, the transition weights of its forwarder are set to be remain in state `enabled`, and transition `enable` is invoked. After the forwarder is in state `enabled`, the `end` transition is executed, and the stored published tokens are processed by method `waitForCompletion` in Paho. That method blocks until the message delivery of the corresponding token finishes, or a timeout occurs. After processing all the stored tokens, the sender disconnects from the server, and terminates the forwarder by invoking transition `end` in it.

Table 1: QoS of messages and message arrival.

Message QoS	Message arrival
QoS 0	$P \geq R$
QoS 1	$P \leq R$
QoS 2	$P = R$

The *retain* flag of each message is set to true so that the message is delivered to the receiver even if the server receives the message while the receiver is not connected (Banks and Gupta, 2014). As an MQTT server stores only one message for each topic, every message in this experiment is assigned a unique topic.

Receiver. Similarly to the model of sender, the receiver model (see Figure 3d) creates and launches an instance of forwarder. The receiver then subscribes to the sender's topics. In order not to miss any messages after the receiver disconnects from the server, transition `stop` can be executed only after the sender has disconnected. This dependency is managed by preconditions. The remaining transitions work the same as those of the sender, except that the receiver neither publishes messages nor waits for their completion.

The test oracle is embedded at the end of the `end` transition in the receiver, as an `assert` statement. The actual condition of the assertion varies depending on the QoS of the messages in the experiment as shown in Table 1. In this table, P, R denote the number of the messages published from the sender and the number of the messages the receiver received, respectively. A test case is regarded as failed if this condition is violated, or uncaught exceptions are raised.

4.3 Results

We performed 50 tests for each setting. Each test case took about a hundred milliseconds.

Table 2: Results of the experiments testing Paho.

Sender	Receiver	QoS 0	QoS 1	QoS 2
Stable	Unstable	Success	Success	Success
Unstable	Stable	Timeout	Success	Success
Unstable	Unstable	Timeout	Success	Success

Table 2 shows the summary of the results of these experiments for three types of QoS and three types of stability of the clients. In this table, "Success" or "Timeout" shows the result of each experiment, where "Success" means all the test cases finished satisfying the assertion without uncaught exceptions and "Timeout" means a timeout occurred when the sender disconnected from the server in some test cases.

We describe the timeout in detail. The timeout happens in the method `waitForCompletion` in Paho, called in transition `end` of the sender. This timeout means that the message delivery corresponding to the token has not been completed. The timeout only happened with QoS 0 and an unstable sender. This seems to the be correct behavior, because the message delivery is never completed if the message is lost because of a network error. This result shows that network disconnection was correctly simulated, and it interrupted message delivery.

Moreover, there were some test cases in which the inequalities in Table 1 hold strictly. Namely, $P > R$ and $P < R$ held in some test cases with messages with QoS 0 and QoS 1, respectively. This result implies message loss actually occurred in the experiments with QoS 0, and message re-delivery was actually performed in the experiments with QoS 1.

5 CONCLUSIONS AND FUTURE WORK

We show an approach that applies model-based testing to model-based simulation, in the context of the IoT protocol "MQTT". This was achieved by extending Modbat, a model-based tester based on extended finite-state machines, with capabilities to model features needed in such scenarios succinctly: (1) Time, to simulate physical time, (2) dynamic weight changes, to simulate transient errors, (3) transition invocation, to handle callbacks. We have also introduced packet forwarders, in order to simulate unstable TCP network environments during test execution entirely in software, without modifying the SUT. With packet forwarders, we are able to observe the effect of lost packets in MQTT, and confirm that under higher quality settings, the protocol repeats transmissions until data is delivered.

In the future, we think our framework can serve as a testbed for evaluating the robustness of controllers under adverse conditions ,testing how delays and retransmissions affect real-time behavior.

ACKNOWLEDGMENTS

This work was supported by JSPS *kakenhi* grant 17H01719.

REFERENCES

Alvi, S. A., Shah, G. A., and Mahmood, W. (2015). Energy efficient green routing protocol for Internet of Multimedia Things. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2015 IEEE Tenth Int. Conf. on*, pages 1–6. IEEE.

Artho, C. and Biere, A. (2018). Modbat. `https://people.kth.se/~artho/modbat/`.

Artho, C., Gros, Q., Rousset, G., Banzai, K., Ma, L., Kitamura, T., Hagiya, M., Tanabe, Y., and Yamamoto, M. (2017). Model-based API testing of Apache ZooKeeper. In *2017 IEEE Int. Conf. on Software Testing, Verification and Validation*, pages 288–298. IEEE Computer Society.

Artho, C., Havelund, K., Kumar, R., and Yamagata, Y. (2015). Domain-specific languages with Scala. In *Proc. 17th Int. Conf. on Formal Engineering Methods (ICFEM 2015)*, volume 9407 of *LNCS*, pages 1–16, Paris, France. Springer.

Artho, C. V., Biere, A., Hagiya, M., Platon, E., Seidl, M., Tanabe, Y., and Yamamoto, M. (2013). Modbat: A model-based API tester for event-driven systems. In *Proc. 9th Int. Haifa Verification Conf.*, volume 8244 of *LNCS*, pages 112–128. Springer.

Banks, A. and Gupta, R. (2014). MQTT version 3.1.1. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`.

Beauquier, D. (2003). On probabilistic timed automata. *Theoretical Computer Science*, 292(1):65–84.

Bormann, C. (2014–2016). CoAP — constrained application protocol — overview. `http://coap.technology/`.

Bulychev, P., David, A., Larsen, K. G., Mikučionis, M., Poulsen, D. B., Legay, A., and Wang, Z. (2012). UPPAAL-SMC: Statistical model checking for priced timed automata. *arXiv preprint arXiv:1207.1272*.

Cheng, K. and Krishnakumar, A. S. (1993). Automatic functional test generation using the extended finite state machine model. In *Proc. of the 30th Design Automation Conf. . Dallas, Texas, USA, June 14-18, 1993.*, pages 86–91. ACM Press.

Claessen, K. and Hughes, J. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. 5th ACM SIGPLAN Int. Conf. on Functional Programming*, pages 268–279. ACM.

Dadeau, F., Héam, P.-C., and Kheddam, R. (2011). Mutation-based test generation from security protocols in HLPSL. In *Software Testing, Verification and Validation, 2011 IEEE Fourth Int. Conf. on*, pages 240–248. IEEE.

De Rubertis, A., Mainetti, L., Mighali, V., Patrono, L., Sergi, I., Stefanizzi, M. L., and Pascali, S. (2013). Performance evaluation of end-to-end security protocols in an Internet of Things. In *Software, Telecommunications and Computer Networks (SoftCOM), 2013 21st Int. Conf. on*, pages 1–6. IEEE.

Eclipse Paho Team (2018). Eclipse Paho—MQTT and MQTT-SN software. `http://www.eclipse.org/paho/`.

Grabowski, J., Hogrefe, D., Réthy, G., Schieferdecker, I., Wiles, A., and Willcock, C. (2003). An introduction to the testing and test control notation (TTCN-3). *Computer Networks*, 42(3):375–403.

Heckel, R. and Torrini, P. (2010). Stochastic modelling and simulation of mobile systems. In *Graph transformations and model-driven engineering*, pages 87–101. Springer.

Houimli, M., Kahloul, L., and Benaoun, S. (2017). Formal specification, verification and evaluation of the MQTT protocol in the internet of things. In *Mathematics and Information Technology (ICMIT), 2017 Int. Conf. on*, pages 214–221. IEEE.

Hsu, Y., Shu, G., and Lee, D. (2008). A model-based approach to security flaw detection of network protocol implementations. In *Network Protocols, 2008. ICNP 2008. IEEE Int. Conf. on*, pages 114–123. IEEE.

Hsueh, M., Tsai, T. K., and Iyer, R. K. (1997). Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82.

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

Jing, C., Wang, Z., Shi, X., Yin, X., and Wu, J. (2008). Mutation testing of protocol messages based on extended TTCN-3. In *Advanced Information Networking and Applications, 22nd Int. Conf. on*, pages 667–674. IEEE.

Khan, A., Torrini, P., and Heckel, R. (2009). Model-based simulation of voip network reconfigurations using graph transformation systems. *Electronic Communications of the EASST*, 16.

Kingsbury, K. (2018). Distributed systems safety research. `http://jepsen.io`.

Kosiuczenko, P. and Lajios, G. (2007). Simulation of generalised semi-markov processes based on graph transformation systems. *Electronic Notes in Theoretical Computer Science*, 175(4):73–86.

Kwiatkowska, M., Norman, G., and Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In *Int. Conf. on Computer Aided Verification*, pages 585–591. Springer.

Kwiatkowska, M., Norman, G., and Sproston, J. (2002). Probabilistic model checking of the IEEE 802.11 wireless local area network protocol. In *Process Algebra and Probabilistic Methods: Performance Modeling and Verification*, pages 169–187. Springer.

Lee, S., Kim, H., Hong, D., and Ju, H. (2013). Correlation analysis of MQTT loss and delay according to QoS level. In *The Int. Conf. on Information Networking*, pages 714–717. IEEE Computer Society.

Light, R. A. (2017). Mosquitto: server and client implementation of the MQTT protocol. *Journal of Open Source Software*, 2(13).

Mikk, E., Lakhnech, Y., Petersohn, C., and Siegel, M. (1997). On formal semantics of statecharts as supported by STATEMATE. In *Workshop, Ilkley*, volume 14, page 15.

Mladenov, K., van Winsen, S., and Mavrakis, C. (2017). Formal verification of the implementation of the MQTT protocol in IoT devices. *SNE Master Research Projects 2016–2017*.

Nilsson, R. (2015). ScalaCheck. `https://www.scalacheck.org/`.

Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala*. Artima Inc.

Oehlert, P. (2005). Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62.

Park, J.-C. and Miller, R. E. (1997). Synthesizing protocol specifications from service specifications in timed extended finite state machines. In *Distributed Computing Systems, 1997., Proc. of the 17th Int. Conf. on*, pages 253–260. IEEE.

Paz, A. (1971). *Introduction to Probabilistic automata*. Academic Press, 1 edition.

Stott, D. T., Ries, G. L., Hsueh, M., and Iyer, R. K. (1998). Dependability analysis of a high-speed network using software-implemented fault injection and simulated fault injection. *IEEE Trans. Computers*, 47(1):108–119.

Tappler, M., Aichernig, B., and Bloem, R. (2017). Model-based testing IoT communication via active automata learning. In *10th IEEE Int. Conf. on Software Testing, Verification and Validation*, pages 276–287. IEEE.

Testware (2018). Project IoT. `http://www.iot-t.de/en/testware/`.

Thangavel, D., Ma, X., Valera, A. C., Tan, H., and Tan, C. K. (2014). Performance evaluation of MQTT and coap via a common middleware. In *2014 IEEE Ninth Int. Conf. on Intelligent Sensors, Sensor Networks and Information Processing*, pages 1–6. IEEE.

Torrini, P., Heckel, R., and Ráth, I. (2010). Stochastic simulation of graph transformation systems. In *Int. Conf. on Fundamental Approaches to Software Engineering*, pages 154–157. Springer.

Utting, M., Pretschner, A., and Legeard, B. (2012). A taxonomy of model-based testing approaches. *Softw. Test., Verif. Reliab.*, 22(5):297–312.

Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., and Nachmanson, L. (2008). Model-based testing of object-oriented reactive systems with Spec Explorer. In *Formal Methods and Testing 2008*, volume 4949 of *LNCS*, pages 39–76. Springer.

Zhang, Z., Wen, Q.-Y., and Tang, W. (2012). An efficient mutation-based fuzz testing approach for detecting flaws of network protocol. In *Computer Science & Service System, 2012 Int. Conf. on*, pages 814–817. IEEE.

Ziade, H., Ayoubi, R. A., and Velazco, R. (2004). A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1(2):171–186.