Model-based Testing Simulating Unstable Networks and Devices for IoT Software
IoT ソフトウェアのための不安定なネットワークとデバイスをシミュレートするモデルベーステスト

by

Jun Yoneyama
米山惇

A Master's Thesis
修士論文

Submitted to
the Graduate School of the University of Tokyo
on February 28, 2018
in Partial Fulfillment of the Requirements
for the Degree of Master of Information Science and Technology
in Computer Science

Thesis Supervisor: Masami Hagiya　萩谷昌己
Professor of Computer Science

ABSTRACT

IoT devices sometimes operate under unstable network environments because of restrictions such as location or size of the devices. In addition, while many devices work in IoT environments, there is a possibility that multiple devices break down at the same time. In order to test whether programs work correctly even under various unstable environments, model-based testing, a testing method which extracts diverse test cases automatically and performs automated tests, is considered to be effective. However, how to describe many devices running asynchronously, and how to implement device failures or unstable network environments in automated testing, are unclear.

In order to solve these problems, this paper proposes extended methods of model-based testing suitable for programs working in IoT environments. First, the test method is extended to improve expressiveness of test models. This extension enables describing unstable systems by designating concerns such as error rates or the magnitude of network delay as statistics. Second, the concept of time is introduced to model-based testing for simulating and testing devices running in real time. Third, packet forwarders are inserted in network communication paths so that they can be controlled by the test tool. This method enables performing tests while causing network disconnection or delay artificially without modifying the programs. Moreover, the test tool is extended by abstracting these three extensions so that it can be easily used when test models are written.

In order to show usefulness of these methods, tests of a server and a client library of MQTT, a transport protocol designed for IoT environments, with Modbat, a model-based test tool, are performed. Moreover, simple applications using an MQTT server and MQTT clients are created and tests of them are performed.

論文要旨

IoT 向けデバイスは, デバイスの位置, 大きさなどの制約により, 不安定なネットワーク環境で稼働する場合がある. 更に, IoT 環境では多数のデバイスが動作するが, 外的な要因により, 複数のデバイスが同時に故障する可能性がある. プログラムがさまざまな不安定な状況でも正常に動作することをテストするためには, 多様なテストケースを機械的に抽出して自動テストを行う手法であるモデルベーステストが有効であると考えられる. しかし, 非同期的に動作する多数のデバイスをどのように記述すればよいか, また, デバイスの故障や不安定なネットワークを自動テストでどのように実現するかは明らかでない.

これらの問題を解決するため, 本論文では, モデルベーステストの手法を拡張し, IoT 環境で動作するプログラムに適したテストを行う手法を提案する. 第一に, デバイスの数, 故障率, ネットワークの遅延の大きさなどの関心事を統計量として指定して, それに従ってデバイスや通信路の故障が発生した場合のテストを簡潔に記述できるようにする. 第二に, モデルベーステストに時間の概念を導入し, 実時間上で動作するデバイス群のシミュレート及びテストを行えるようにする. 第三に, プログラム間のネットワーク通信路にパケットフォワーダーを挿入することにより, ネットワークの切断や遅延をシミュレートし, テストツールから制御できるようにする. この手法では, 検証対象のプログラムを変更せずに, 不安定なネットワークをシミュレートしながらテストを行うことができる. 更に, この三点の拡張を抽象化し, テストを書く際に簡単に利用できるようテストツールの拡張を行う.

これらの手法の有効性を示すため, モデルベーステストツール Modbat を用いて, IoT 環境向けに設計された通信プロトコルである MQTT のサーバー及びクライアントライブラリのテストを行った. 更に, MQTT サーバー・クライアントを利用する簡単なアプリケーションを作成し, アプリケーション単位でのテストを行った.

Acknowledgements

We would like to thank Masami Hagiya for a lot of useful advice and supervision. We would also like to appreciate Cyrille Artho and Yoshinori Tanabe's advice.

Contents

Chapter 1

Introduction

In this thesis, we extend methods of model-based testing for testing IoT systems efficiently.

1.1 Background

In IoT environments, many devices are connected to the Internet. They communicate each other via the Internet to work together. However, these devices have some restrictions in IoT environments.

First, communication methods of devices are limited. In most cases, IoT devices are connected to the Internet wirelessly. However, since device location is limited, devices may be far from routers or base stations. Moreover, some kinds of devices may move around. Because of this restriction, IoT devices need to work under unstable radio condition.

Second, device size is limited because there may not be enough space for IoT devices. Clearly, only small antennas can be installed on a small device. This restriction also makes the network connection of devices unstable. In addition, a small device can mount only small battery packs, which increases risks that the device battery run out.

Because of these restrictions, IoT devices are required to work under unstable network environments and unstable device condition.

1.2 Our Research

In an IoT system, many devices work together communicating one another. In order to find bugs in programs working on these devices, the programs need to be tested. Many programs running asynchronously may produce many execution paths, therefore model-based testing is considered to be suitable for such situations. However, it is difficult to test these programs with usual model-based testing. For example, it is easy to describe behavior of each device in model-based testing, but it is difficult to describe the whole behavior of the system. In addition, methods to simulate device failures or unstable network environments in automated testing are not established.

This research proposes extensions for model-based testing to perform testing suitable for programs working in IoT environments. Our methods consist of mainly two approaches. First approach is to extend the definition of model-based testing in order to describe IoT systems as test models. This approach consists of three extensions. First extension introduces *timed extended finite state machines*, which enable describing systems working in real time. Second extension, *dynamic weight change* enables describing devices whose error rates may

be changed. Third extension, *transition invocation* simplifies model description of devices which handle errors with callback functions, or devices whose states are changed by the environment forcibly. Second approach is to simulate unstable network environments during test execution. It introduces a mechanism to simulate unstable network line by software. We implemented these extensions in Modbat, a model-based test tool.

Moreover, we created test models of IoT systems using these extensions and performed tests of the models with Modbat.

1.3 Outline

This thesis is structured as follows. Chapter 2 summarizes the related work. Chapter 3 describes the background knowledge for this research. Chapter 4 describes the definitions and implementations of our extensions. The methods and the results of the experiments are described in Chapter 5. Finally, Chapter 6 concludes this thesis and Chapter 7 discusses the future work.

Chapter 2

Related Work

This chapter describes relevant work. Since our study proposes methods to test software on IoT environments using a model-based test tool and fault injection, we refer studies on model-based testing, fault injection, and IoT testing.

2.1 Model-based Testing

Testing is a kind of method to test programs by executing a part of the programs and comparing expected output and actual output. Model-based testing is a variant of testing in which abstract models of the software are described to generate test cases automatically [25]. There are some test tools for model-based testing, such as QuickCheck [11], ScalaCheck [19] and Modbat [7].

In concurrent systems, the output of the system may be non-deterministic, and there is an approach to verify the output by enumerating possible outputs [5].

2.2 Fault Injection

Fault injection is a technique to inject errors into the system artificially in order to test the robustness of the system [21], [14]. This technique is also applied to software testing as software implemented fault injection (SWIFI). There are tools for SWIFI, such as FERRARI [15], DOCTOR [13] and MODIFI [23].

There is a study that evaluates effects of network error on a Myrinet network interface hardware by using simulated fault injection and software implemented fault injection [22]. Both methods in this study modify the system under test (SUT) to inject faults. Our study differs from this in that our method injects faults with software, without modifying the SUT.

2.3 IoT and Testing

There are transport protocols designed for IoT devices such as MQTT [8] and CoAP [9]. In these protocols, multiple clients communicate one another via a server.

There is an approach to verify MQTT implementations formally [18]. In this study, the specification is described in a formal language TTCN-3 [12], and MQTT servers are tested according to the specification, by sending messages and receiving responses from them. They tested three MQTT server implementations and found that all of them produced responses violating the protocol specification.

In a study evaluating the performance of MQTT and CoAP, unstable network environments are realized by using running a WAN emulator in another PC [24]. This study evaluates correlation between network instability and performance,

such as the size of network delay and packet loss rate. Our approach differs from this study in that no extra equipment is required, and that the network environment can be easily controlled by the software during a test. However, this study may be superior to our study in that this method simulates network errors more faithfully than our method.

There is also a study that analyzes loss and delay of MQTT messages in actual wireless network environment by capturing TCP packets [17]. This study analyzes correlation between the size of message payload and message loss, on wired network and wireless network. This method enables testing in actual environments, but our method can control network errors more finely by software.

Our approach differs from these studies in that network disconnection is simulated in a situation close to the actual environment, that MQTT clients are also tested, and that the whole system can be tested.

Chapter 3

Preliminaries

This thesis extends the method of model-based testing, and performs tests of MQTT implementations. Model-based testing is a kind of test method for software testing. MQTT is a transport protocol designed for IoT environments. This chapter describes the background knowledge about model-based testing and MQTT.

3.1 Model-based Testing

Software testing is a method to find bugs in software by executing some parts of the software. The target software for the test is called system under test (SUT). Usually, one test case consists of a pair of an input and an expected output, and a test is performed by comparing the actual output and the expected output.

Model-based testing is one of the methods of software testing. In this method, test writer defines a model abstracting behavior of the SUT as an abstract model. This method enables generating multiple test cases from the model.

3.1.1 Modbat

Modbat is one of test tools for model-based testing. [7, 4] This tool is mainly designed for testing an SUT with an application programming interface (API). A model in Modbat is expressed as an extended finite state machine (EFSM) in a domain-specific language based on Scala [16, 20].

Modbat is implemented in Scala and is able to test APIs in Java and Scala.

3.1.2 Extended Finite State Machines

Definition 3.1. An extended finite state machine is defined as a 7-tuple $M = (I, O, S, D, F, U, T)$, where

- S is a set of symbolic 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 such that $f_i \colon D \to \{0, 1\}$,
- U is a set of *update functions* u_i such that $u_i \colon D \to D$, and
- T is a transition relation such that $T \subseteq (S \times F \times I) \times (S \times U \times O)$ [10].

Here D stands for the internal state of the model.
And the state of the model at a specific time is expressed as follows.

Definition 3.2. A state of the EFSM M is defined as $(s, \boldsymbol{x})$, where $s \in S$ and $\boldsymbol{x} \in D$.

We denote a transition $((s_1, f, i), (s_2, u, o)) \in T$ as $(s_1, f, i) \to (s_2, u, o)$, where $s_1, s_2 \in S$, $f \in F$, $i \in I$, $u \in U$ and $o \in O$. Intuitively, a transition $(s_1, f, i) \to (s_2, u, o)$ means that when the system is in the symbolic state s_1, its internal state $\boldsymbol{x}$ satisfies $f(\boldsymbol{x}) = 1$ and the input i is given, the SUT moves to the symbolic state s_2 with generating output o and updating the internal state $\boldsymbol{x} := u(\boldsymbol{x})$ [10].

3.2 MQTT

MQTT is a transport protocol designed for IoT environments. [8] This protocol has some characteristics suitable for devices with unstable network environments and limited power sources. This protocol runs over TCP/IP or another transport layer protocol which supports ordered, lossless and bidirectional communication. [8] We assume TCP/IP connection and use terms of TCP/IP in this paper. We also focus on MQTT version 3.1.1.

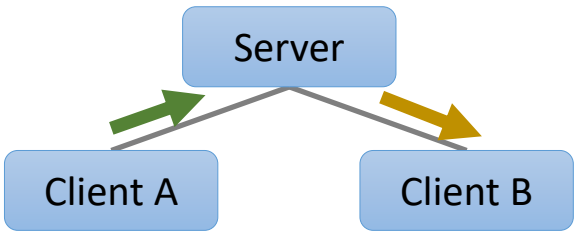

Figure 3.1: An example system communicating with MQTT

MQTT is a *publish/subscribe pattern* protocol in which multiple clients exchange messages via a server. A server is sometimes called a broker in this protocol. Each message has a payload, a *topic name* and a value named *Quality of Service* (QoS). This protocol can guarantee message arrival even if devices are working in unstable network environments that TCP connection may be lost. A topic name is a string used to control delivery of messages.

QoS is used to guarantee message arrival. MQTT defines three types of QoS, namely 0 to 2 as follows.

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

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

QoS 2 This message is delivered "exactly once". No message lose or redundant delivery is allowed. [8]

3.2.1 Topic Name and Topic Filter

A topic name is a string sequence separated into hierarchies by slashes. A client can *subscribe topic filters*, which represent a set of topics. When the server receives a message from a client, the server forwards the message to clients of which topic filters match the topic name of the message.

The structure of a topic filter is similar to that of a topic name, but the usage of wildcards is allowed in a topic filter.

3.2.2 Interactions in MQTT

Here we describe the overall behavior of interactions between a server and a client.

Connect

When a server is running, it is waiting for a TCP connection at a particular port. Connection process begins with TCP connection request from the client. After the connection is established, the client sends a `CONNECT` packet to the server. The `CONNECT` packet includes a string *client identifier* and several flags.

We describe *clean session* flag here. Clean session is a one-bit value included in the `CONNECT` packet to define treatment of the *session* state. A Session is an interaction between the server and the client which stores various states between them. If the clean session flag is set to 1 in the `CONNECT` packet, the server discards all information about the client and starts entirely new session then. If it is set to 0, the session is regarded as the continuing session from the previous connection with the client.

After receiving the `CONNECT` packet, the server responds with a `CONNACK` packet, indicating acceptance or rejection.

Subscribe

The client subscribes topics to receive messages from the server with a `SUBSCRIBE` packet. A `SUBSCRIBE` packet consists of a list of pairs of a topic filter and a requested QoS. A pair of a topic filter and a QoS indicates subscribing messages with topics matching the filter with at most the desired QoS.

The server responds to the client with a `SUBACK` packet, which shows success or failure for each topic filter.

The client can quit subscription with an `UNSUBSCRIBE` packet, and the server responds to it with an `UNSUBACK` packet.

Publish

A message is published from a client to a server, or a server to a client. Packet interaction is the same in both directions. Here we distinguish a client and a server with the words a sender and a receiver.

In a publish process, the sender sends a `PUBLISH` packet to the receiver first. The packet includes a topic, a QoS, a retain flag and a payload. The response of the receiver varies depending on the QoS.

QoS 0 The receiver makes no response.

QoS 1 The receiver responds with a `PUBACK` packet. The sender sends the `PUBLISH` packet again if the network connection is lost without receiving the `PUBACK` packet.

QoS 2 The receiver responds with a `PUBREC` packet. If the sender does not receive the `PUBREC` packet, it sends the `PUBLISH` packet again. After receiving the `PUBREC` packet the sender sends a `PUBREL` packet, and again the receiver responds with a `PUBCOMP` packet. The sender also sends the `PUBREC` packet again if the `PUBCOMP` packet does not reach the sender.

A PUBLISH packet in QoS 1 or QoS 2 also has a value packet identifier, and the responses for the packet have the same packet identifier to indicate the relation to the message.

A retain flag is one-bit value in a PUBLISH packet. If the flag is set to 1 in a PUBLISH packet from the client, the server stores the message for future subscriptions. One message is stored for each topic, therefore the arrival of another message with the same topic and retain flag 1 overwrites the stored message.

Disconnect

When the client disconnects from the server, the client sends a DISCONNECT packet. The server makes no response and closes the TCP connection. If the connection started with setting clean session to 0, the server stores the session state with the client, including its subscriptions and unacknowledged messages in QoS 1 or QoS 2.

3.2.3 Implementations

The specification of MQTT is published, and there are many open source implementations for MQTT servers and clients.

Eclipse Mosquitto is an open source broker implementation for the MQTT protocol versions 3.1 and 3.1.1 [2]. Mosquitto also contains a C and C++ client library. The broker is mainly written in C language.

Eclipse Paho is an open source project including implementations of MQTT libraries for many programming languages [1]. Each library provides APIs for MQTT communications, such as connecting, publish messages, subscribing and disconnecting.

Both Eclipse Mosquitto and Eclipse Paho are parts of iot.eclipse.org projects.

Chapter 4

Methods

We propose extensions for model-based testing and have implemented them in Modbat. All of those extensions are designed for testing IoT devices, but they may be also useful for testing other software. In this chapter, we describe the purpose, the definition and the implementation of each extension.

Section 4.1, 4.2 and 4.3 describe extensions for model-based testing for improving description of models. These extensions enable describing models which are not describable in original model-based testing, or simplify model description. Each of these sections shows the formal definition and the implementation in Modbat of each extension. Section 4.4 describes an extension for test execution, which enables testing while simulating unstable network environments.

4.1 Timed Extended Finite State Machines

In an EFSM, the transitions are decided by the symbolic state $s \in S$, the internal state $\boldsymbol{x} \in D$ and the input $i \in I$. However, actual systems sometimes affected by the time, therefore it is natural to extend an EFSM to utilize the time to decide whether a transition is able to be executed.

Though it is possible to extend enabling functions to take the time for an additional argument, we need to take care of difference between the internal state condition and the time condition. Namely, if there is no possible transition because of the enabling functions, the model will never have an executable transition. On the other hand, if there is no possible transition because of the time conditions, it is possible to continue transition in the future, with some time conditions satisfied.

We designed an extension for an EFSM to deal with the time in a limited way so that the models are simply described and easily confirmed their terminating condition. This extension defines timed extended finite state machines (TEFSMs), and a transition in a TEFSM may suspend some transitions in the model for a specific period. While some transitions are suspended, the other transitions are able to be executed.

4.1.1 Definition

Definition 4.1. A timed extended finite state machine (TEFSM) is defined as an 8-tuple $M = (I, O, S, D, F, U, \Pi, T)$, where

- I, O, S, D, F, U are the same as those of Definition 3.1,
- Π is a set of *suspending functions* π_i such that $\pi_i \colon T \to \mathbb{R}_{\geq 0}$, and
- T is a transition relation such that $T \subseteq (S \times F \times I) \times (S \times U \times \Pi \times \mathbb{R}_{\geq 0} \times O)$.

Definition 4.2. A state of the TEFSM M is defined as a 4-tuple $(s, \boldsymbol{x}, p, \tau)$, where $s \in S$, $\boldsymbol{x} \in D$, $p\colon T \to \mathbb{R}_{\geq 0}$ and $\tau \in \mathbb{R}_{\geq 0}$.

Here $p(t) \in \mathbb{R}_{\geq 0}$ represents the suspending time for each transition $t \in T$, and τ represents the current time of the test. A test starts from the state with $p(t) = 0$ for all $t \in T$ and $\tau = 0$. We describe additional variables in $t = (s_1, f, i) \to (s_2, u, \pi, e, o)$. The variable $e \in \mathbb{R}_{\geq 0}$ represents the execution time of t, namely, when the transition t is executed, the time advances by e.

When the system is in the state $(s, \boldsymbol{x}, p, \tau)$, a transition $t = (s_1, f, i) \to (s_2, u, \pi, e, o)$ is executable if and only if $s = s_1$, $f(\boldsymbol{x}) = 1$ and $p(t) \leq \tau$. And executing the transition t updates the state by

$$s := s_2 \tag{4.1}$$
$$\boldsymbol{x} := u(\boldsymbol{x}) \tag{4.2}$$
$$p := p' \tag{4.3}$$
$$\text{where} \quad p'(t') = \tau + e + \pi(t') \tag{4.4}$$
$$\tau := \tau + e. \tag{4.5}$$

4.1.2 Implementation

We implemented this extension in Modbat as an additional method for Modbat. This implementation is designed for models executing multiple model instances in parallel. Each model instance has a boolean flag `staying`, which means all the transitions in the model instance are suspended if it is set to true.

The value e in a transition in a TEFSM models the execution time of the transition, but the execution time may vary in actual test executions. In this implementation, appending `stay` method to a transition function can assign the suspending time to the transition in the model. This method is overloaded in two ways, with one integer value or a pair of two integer values. After a transition function with `stay (x, y)` attached is executed, a random integer `z` larger than `x` (inclusive) and smaller than `y` (exclusive) is chosen. Then the model instance suspends for `z` milliseconds and other model instances keep running. Internally, the variable `staying` is set to true then, and a timer thread is started. The timer thread sleeps for `z` milliseconds and then sets the `stay` variable of the model instance to false. Here `z` corresponds to the suspending time $p(t)$ in the TEFSM, but this implementation extends the suspending time non-deterministically. The method with one integer argument is a syntax sugar of that with a pair of two integers argument, namely, `stay x` is equivalent to `stay (x, x)`. These methods can be used like an attribute of a transition, by using method chain notation of Scala (Figure 4.1).

At each moment Modbat tries to execute a transition, it enumerates all possible transitions and terminates the test execution if there is no transition satisfying its enabling function. However, if there is no executable transition but there are transitions in staying model instances, it is expected for Modbat to wait until one of the staying model instances finishes staying. This functionality is implemented in Modbat by using a unique object `stayLock` and methods for handling threads in Java, `wait` and `notify`. When Modbat enumerates all possible transitions and find only transitions in staying model instances, the main thread calls `wait` method with the object `stayLock`. On the other hand, the timer thread calls `notify` with `stayLock` after sleeping and setting `staying` to true. Then the main thread is resumed, and the test execution continues running.

```
class A extends Model {
  // suspended for 100 ms after execution
  "a" -> "b" := {
    // some actions
  } label "x" stay 100

  // suspended for a specific duration between 150 ms and 200 ms
  "b" -> "c" := {
    // some actions
  } label "y" stay (150, 200)

  "c" -> "d" := {
    // some actions
  } label "z"
}
```

Figure 4.1: A Modbat model using two types of stay methods

```
class A extends Model {
  var lastTransition: Long = _
  // suspended for 100 ms after execution
  "a" -> "b" := {
    // some actions
    lastTransition = System.currentTimeMillis()
  } label "x"

  // suspended for a specific duration between 150 ms and 200 ms
  "b" -> "c" := {
    require(System.currentTimeMillis() - lastTransition >= 100)
    // some actions
    lastTransition = System.currentTimeMillis()
  } label "y"

  "c" -> "d" := {
    require(System.currentTimeMillis() - lastTransition >=
            choose(150, 200))
    // some actions
  } label "z"
}
```

Figure 4.2: A Modbat model with similar behavior to that of Figure 4.1 without using stay method

Figure 4.1 shows an example model to use two types of stay methods. The model in Figure 4.2 reproduces similar behavior without using `stay` method, but its description is complex. The code will be more complex if the model has more transitions. Moreover, if the require statement is not satisfied and there is no other executable transition, the test execution terminates then, even though the transitions can be executable in the future.

4.2 Dynamic Weight Change

An EFSM in Modbat behaves probabilistically in actual test executions according to the *weights* of the transitions. Each transition has a real number named weight, and the probability that the transition is chosen is proportional to its weight. This functionality is useful for simulating non-deterministic behavior of the SUT and for generating various test cases.

In this paper, we propose extension for this functionality so that the weights can be modified dynamically during a test execution. This feature cannot be realized in the original model-based testing. This extension enables simulating situations that many devices break probabilistically, but the error rates change depending on the environment. Such situations can be easily implemented by creating a model that models the environment and controls the error rates of the device models.

4.2.1 Definition

Before defining our proposed extension, we formalize this probabilistic behavior as follows.

Definition 4.3. A probabilistic extended finite state machine (PEFSM) is defined as an 8-tuple $M = (I, O, S, D, F, U, T, w)$, where

- I, O, S, D, F, U, T are the same as those of Definition 3.1 respectively, and
- w is a *weight function* $w\colon T \to \mathbb{R}_{\geq 0}$, where $\mathbb{R}_{\geq 0}$ is the set of non-negative real numbers.

The state of this model is expressed as $(s, \boldsymbol{x}) \in S \times D$, which is the same as that of EFSM.

The transition probability is decided by the weight function. First, the weight of a transition $t = ((s_1, f, i) \to (s_2, u, o)) \in T$ is defined by $w(t)$. Second, a set of possible transition $T_{s,\boldsymbol{x},i}$ for each model state $(s, \boldsymbol{x}) \in S \times D$ and input $i \in I$ is defined as follows.

$$T_{s,\boldsymbol{x},i} = \{((s_1, f, j) \to (s_2, u, o)) \in T \mid s = s_1, f(\boldsymbol{x}) = 1, i = j\} \tag{4.6}$$

Then the probability that $t \in T_{s,\boldsymbol{x},i}$ is chosen is decided proportionately to $w(t)$, indeed,

$$\frac{w(t)}{\sum_{t' \in T_{s,\boldsymbol{x},i}} w(t')}. \tag{4.7}$$

In Modbat, weights are expressed as a variable of type double attached to a transition.

We extend PEFSMs to enable updating transition weights dynamically.

Definition 4.4. A dynamic probabilistic extended finite state machine (DPEFSM) is defined as a 9-tuple $M = (I, O, S, D, F, U, \Omega, T, w_0)$, where

- I, O, S, D, F, U are the same as those of Definition 3.1 respectively,
- Ω is a set of *weight update functions* ω_i such that $\omega_i\colon T \to \mathbb{R}_{\geq 0} \cup \{*\}$, where $*$ is a fresh symbol,
- T is a transition relation such that $T \subseteq (S \times F \times I) \times (S \times U \times W \times O)$, and

```
class Environment extends Model {
  var device: Device = _
  "init" -> "stable" := {
    device = new Device
    launch(device)
  }
  "stable" -> "unstable" := {
    device.setWeight("run", 0.5)
    device.setWeight("break", 0.5)
  }
  "unstable" -> "stable" := {
    device.setWeight("run", 0.9)
    device.setWeight("break", 0.1)
  }
}
```

```
class Device extends Model {
  "running" -> "running" :=
    skip label "run" weight 0.9
  "running" -> "broken" := {
    // some actions
  } label "break" weight 0.1
  "broken" -> "broken" :=
    skip label "break" weight 0.1
  "broken" -> "running" := {
    // some actions
  } label "run" weight 0.9
}
```

Figure 4.3: Modbat models of the environment (left) and a device (right) using setWeight

- w_0 is an initial weight function $w_0 \colon T \to \mathbb{R}_{\geq 0}$.

Definition 4.5. A state of the DPEFSM M is defined as a triple $(s, \boldsymbol{x}, w)$, where $s \in S$, $\boldsymbol{x} \in D$ and $w \colon T \to \mathbb{R}_{\geq 0}$.

When the state of this model is $(s, \boldsymbol{x}, w)$, a transition $((s_1, f, i), (s_2, u, \omega, o)) \in T$ is executable if and only if $s = s1$ and $f(\boldsymbol{x}) = 1$. The weight of such a transition t is calculated by $w(t)$ for this state, and the probability that t is chosen is decided by the same way as in a PEFSM.

In this model, when a transition $t = ((s_1, f, i), (s_2, u, \omega, o)) \in T$ is executed with the state $(s, \boldsymbol{x}, w)$, the state is updated by

$$s := s_2 \tag{4.8}$$

$$\boldsymbol{x} := u(\boldsymbol{x}) \tag{4.9}$$

$$w := w' \tag{4.10}$$

$$\text{where} \quad w'(t') = \begin{cases} w(t') & \text{if } \omega(t') = * \\ \omega(t') & \text{otherwise} \end{cases}. \tag{4.11}$$

This update means the weights of some of the transitions is updated, and the other weights are leaved unchanged.

4.2.2 Implementation

In Modbat, a string named label can be attached to a transition. The extension is implemented by adding `setWeight` method in `Model` class. This method receives a string l and a double value w, and sets the weights of the transitions whose labels correspond to l, to w.

Figure 4.3 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 repeats going to "running" state at 90% probability and to "broken" state at 10% probability. When the model of the environment goes to "unstable" state, the weights of the transitions in the device model are updated, and the model goes to "running" state at 50% probability and to "broken" state at 50% probability.

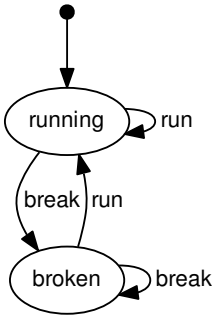

Figure 4.4: The diagram of the model of the device in Figure 4.3

4.3 Transition Invocation

Model-based testing is good at describing actions of the SUTs, but it is difficult to describe passive reactions of the SUTs. For example, it is common to use callback functions in the software, and it is natural that the callback functions change the symbolic state of the software. Furthermore, when multiple models are running in parallel, it is possible for a model to control the symbolic state of another model.

This functionality can be realized in original EFSMs by modifying the internal state by callback functions and restricting transitions by the enabling functions. However, this implementation requires additional variables and enabling functions, and makes the model more complex.

In order to solve this problem, we propose an extension for model-based testing that enables invoking transition functions directly by callback functions or other models.

4.3.1 Definition

Definition 4.6. An invokable extended finite state machine (IEFSM) is defined as a 9-tuple $M = (I, O, S, D, F, U, L, \lambda, T)$, where

- I, O, S, D, F, U are the same as those of Definition 3.1,
- L is a set of label symbols,
- λ is a labeling function, such that $\lambda\colon L \to T$, and
- T is a transition relation such that $T \subseteq (S \times F \times I) \times (S \times U \times L^* \times O)$, where L^* is the set of finite sequences of label symbols.

Definition 4.7. A state of IEFSM M is defined as a triple $(s, \boldsymbol{x}, q)$, where $s \in S$, $\boldsymbol{x} \in D$, and q is a queue of the elements of L.

Intuitively, the queue of the labels q represents a queue of next transitions. If q is not empty, the front transition of q is executed if possible, instead of choosing possible transitions from T. In addition, a transition function can push elements to q when it is executed.

Formally, when the system is in the state $(s, \boldsymbol{x}, q)$ and q is not empty, let $l \in L$ be the front element of q, and let t be $t = (s_1, f, i) \to (s_2, u, (l_1, \ldots, l_n), o) = \lambda(l)$. At this moment l is popped out from q. If t satisfies the condition $s = s_1$, $f(\boldsymbol{x}) = 1$ and the input i is given, t is executed. Otherwise, a transition is looked up from q again in the same way. If q is empty, the executed transition is chosen by the same method as that of EFSMs.

```
class Device extends Model {
  "running" -> "running" := {
    // some actions
  } label "action"
  "running" -> "broken" := {
  } label "error" weight 0.0
  "broken" -> "running" := {
    // handle error
  } label "handle"
  def callbackOnError(): Unit = {
    invokeTransition("error")
  }
}
```

Figure 4.5: An example model using invokeTransition

```
class Device extends Model {
  var hasError: Boolean = false
  "running" -> "running" := {
    require(!hasError)
    // some actions
  } label "action"
  "running" -> "broken" := {
    require(hasError)
  } label "error"
  "broken" -> "running" := {
    // handle error
    hasError = false
  } label "handle"
  def callbackOnError(): Unit = {
    hasError = true
  }
}
```

Figure 4.6: A model with similar behavior without invokeTransition

When a transition $t = (s_1, f, i) \to (s_2, u, (l_1, \ldots, l_n), o)$ is executed, the labels $l_1, \ldots, l_n$ are pushed at the back of q.

4.3.2 Implementation

We implemented a mechanism to invoke transition during testing in Modbat by a method similar to the definition of IEFSMs. We introduced a queue of labels in each test execution, and implemented a method `invokeTransition` in the model class. The transition labels can be pushed to the queue by calling this method from transition functions or callback functions.

The code in Figure 4.5 shows a part of example model using `invokeTransition` in a callback function. Concrete actions and registration of the callback function are not shown in this code. We assume that the callback function `callbackOnError` is called when an error occurs. The transition labeled "error" has weight 0.0, therefore this transition is not executed normally. When an error happens while the model is running, the callback function is called and the label "error" is pushed to the queue. Then at the next time a transition function is executed, the transition function with the label "error" is chosen to be executed.

On the other hand, the code in Figure 4.6 shows a model with similar behavior to that in Figure 4.5, which is implemented without using `invokeTransition`. In this code, the error condition is managed with a boolean variable `hasError`. When the model instance is in "running" state, if `hasError` is set to false then the transition labeled "action" is executed, otherwise the transition labeled "error" is executed. Therefore similar behavior can be reproduced in the original Modbat. However, even if `hasError` is set to true, other transitions in other model instances may be executed at this moment. In addition, it is necessary to take care of data races in setting the value of `hasError`, because it is common that callback functions are executed in other threads. With transition invocation, on the other hand, the test tool manages race conditions, therefore the test writer needs to pay less attention to data races.

4.4 Packet Forwarder

In order to test whether programs work under unstable network environments, it is necessary to simulate network environment. Such methods are generally called *fault injection*, which injects error in the SUT or the environment artificially [14].

We propose a method to simulate network errors and delays in software testing. This method is applicable for software in which servers and clients communicate one another via TCP/IP.

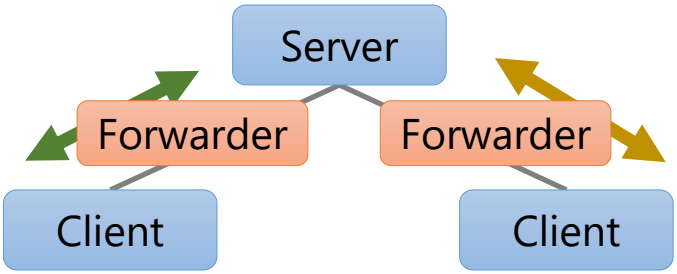

Figure 4.7: A diagram of a system using packet forwarders

This method simulates an unstable network line by inserting a mechanism named *packet forwarder* between a server and a client as shown in Figure 4.7. The packet forwarder normally forwards packets to both directions, from the server to the client and the client from the server. And network errors and delays can be caused artificially by controlling the forwarder. The order of the packets is not modified in this method, because packet order is guaranteed in the TCP specification.

In this method, the program of the SUT is not modified at all, which prevents additional bugs from being mixed into the SUT.

4.4.1 Implementation

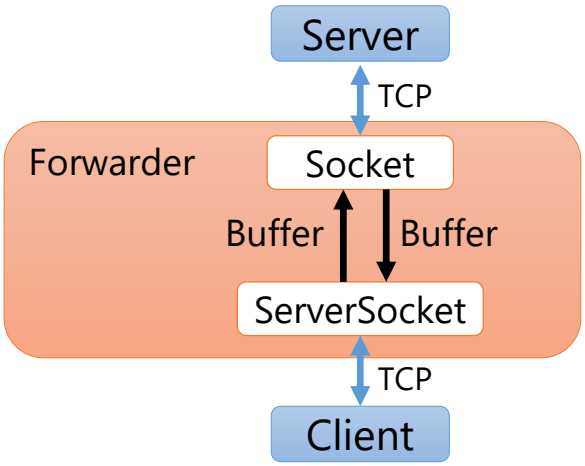

Figure 4.8: A diagram of the contents of a packet forwarders

Figure 4.8 shows the summary of implementation of a packet forwarder in Scala. In this figure, the `Socket` and the `ServerSocket` are instances of the classes `java.net.Socket` and `java.net.ServerSocket` respectively. Informally speaking, the `Socket` acts like the client to the server, and the `ServerSocket` acts like the server to the client. The forwarder also has two buffers, corresponding to

both directions of communication between the server and the client. Each buffer stores the contents of the packet from one socket and sends the contents to the other socket. This forwarding is repeated in an independent thread.

A packet forwarder is launched designating the URL and the port of the server, and the port to accept a connection from the client. When the forwarder is started, it connects to the server with the `Socket` and starts waiting for a connection from a client. And it starts forwarding messages when a client connects to the `ServerSocket` in the forwarder.

When a connection loss is simulated, both the `Socket` and the `ServerSocket` call `close` method to each TCP connection. And when a network delay is simulated, the threads forwarding data between the sockets sleep for a particular time duration. Therefore the network delays in both direction can be controlled independently.

It is useful to implement a packet forwarder as a model instance in Modbat and control the forwarder using model states and transition functions.

Chapter 5

Evaluation

In order to show the effectiveness of the methods described in Chapter 4, we made some models and performed tests of the models with Modbat.

Experiments in Section 5.1 test whether an MQTT server and an MQTT client library work correctly under unstable network environments, especially focusing on re-delivery of messages. These experiments evaluate the effectiveness of packet forwarders and transition invocation.

Experiments in Section 5.2 test a system of a smart house. This system has thermometers and air conditioners communicating with MQTT, and tries to keep the room temperature around the set temperature. These experiments evaluate the effectiveness of extensions for describing model of systems with many devices.

Table 5.1 shows the execution environment of the experiments in this chapter. We implemented our extensions on Modbat version 3.2 and the modified version of Modbat is used in the experiments.

Table 5.1: Execution environment of the experiments

CPU	Intel Core i7-3770 (3.40 GHz)
Memory	16 GB
OS	Ubuntu 16.04
Version of Java	1.8.0_161
Version of Scala	2.11.8
Base version of Modbat	3.2
Version of Mosquitto	1.4.14
Version of Paho	1.2.0

5.1 Test of MQTT Client Library and Server

We conducted experiments to test Mosquitto and Java APIs of Eclipse Paho. We made Modbat models of MQTT clients which call APIs of the library.

5.1.1 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 particular topics.

4. The Sender publishes a message with the topic to the server several times. The Sender also counts the number of the messages published. At this moment, the server is expected to forward the message to the Receiver.

5. The Receiver receives messages from the server and counts the number of the messages it received.

6. The Sender and the Receiver disconnect from the server.

7. The number of the messages published and the number of the messages received are compared.

In actual test executions, multiple test cases are executed sequentially without rebooting the server in order to reduce test execution time. In order to avoid execution of the test cases to conflict one another, the client identifiers of the Sender and the Receiver and the topics in the messages have a common prefix, which is unique to the test case. Concretely, each test case in Modbat is generated from a random seed and the hexadecimal expression of the random seed is used as the prefix.

We conducted experiments to test whether this system works correctly under unstable network environments. In order to simulate unstable network environments, packet forwarders are inserted between each client and the server, and the network lines are controlled. We conducted nine experiments with three types of QoS and three types of the stability of the network lines of the Sender and the Receiver. In each experiment, messages are published in QoS 0, 1 or 2. And the experiments in each QoS are conducted with three different combinations of its Sender and Receiver, which is a stable Sender and an unstable Receiver, an unstable Sender and a stable Receiver, or an unstable Sender and an unstable Receiver. Each of a stable client and an unstable client has a packet forwarder between the client and the server. However, the forwarder of the stable client is always alive during an experiment, but the forwarder of the unstable client sometimes cuts the network connection between the client and the server.

In these experiments, each of the Sender and the Receiver runs an instance of `MqttAsyncClient` class in Paho, which provides non-blocking APIs for MQTT communication. Though the server Mosquitto is running outside the test tool and Modbat calls only APIs in Paho, we cannot simply determine which of the server and the clients has bugs from the result. Therefore both Mosquitto and Paho are considered as the SUTs for these experiments.

5.1.2 Models

Top Level Model

A model for this experiment consists of four types of model instances. Figure 5.1 shows the state machine of the top level model, which is launched at the beginning of the test execution. This model creates model instances of Sender and Receiver, and launches them.

Forwarder

Figure 5.2 shows the state machine of the model of a packet forwarder. This state machine mainly has two states, `enabled` and `disabled`. The weights of the transitions between these two states are modified by the `setWeight` method, so

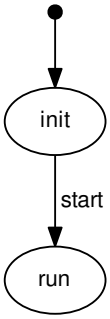

Figure 5.1: The top level model for testing Paho

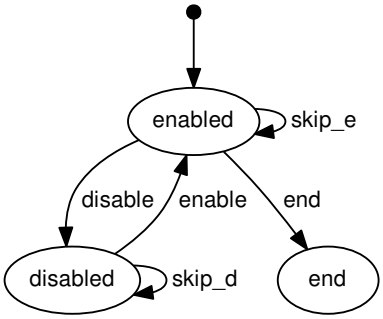

Figure 5.2: A model of Forwarder

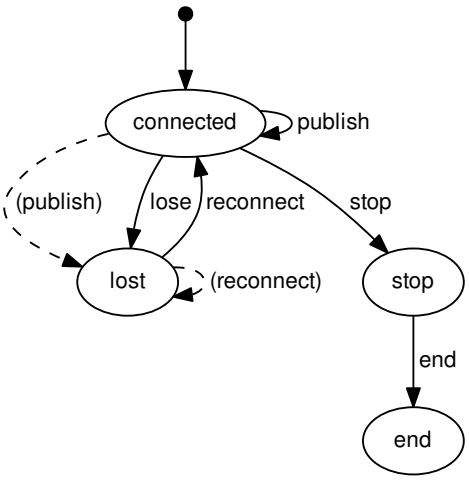

Figure 5.3: A model of Sender

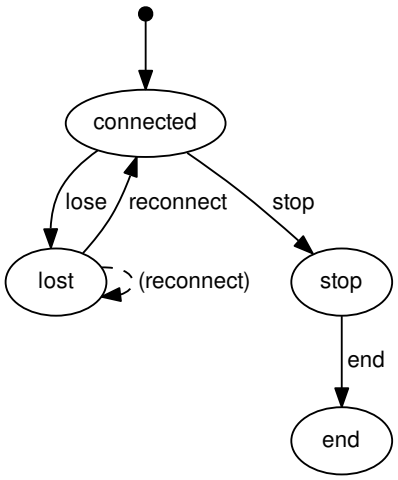

Figure 5.4: A model of Receiver

that the error rate of the Forwarder can be controlled. When the model is in the `enabled` state, the Forwarder repeats forwarding packets in other threads.

There are four transitions between the `enabled` state and `disabled` state, but the two transitions labeled `skip_e` and `skip_d` do nothing. The `disable` transition closes the connections between the server and the Forwarder, and between the client and the forwarder. This transition also terminates the threads forwarding packets. Executing the `enable` transition connects the forwarder to the server and starts waiting for a connection from the client again. The `end` transition terminates waiting for a connection. The weight of this transition is set to 0.0, therefore this transition is not executed unless it is explicitly invoked by `invokeTransition`.

Sender

Figure 5.3 shows the state machine of the Sender model. When an instance of Sender model is launched, it creates an instance of Forwarder and launches it. After that, Sender connects to the server via the Forwarder.

As shown in the figure, the Sender repeats publishing messages when it is at the `connected` state. The publish method in `MqttAsyncClient` returns a *token*, which stores the progress of the message delivery. The model stores all the tokens of publishes a list. If the Sender tries to publish a message while the Forwarder is disabled, publish API in Paho throws an exception and the model goes to the `lost` state, which is shown as a dashed arrow in the figure. This functionality is realized by the `nextIf` method in Modbat. The weight of the transition labeled

`lose` is set to 0.0, therefore this transition is not executed usually. This transition is called by the callback function for `MqttAsyncClient` which is called when the network connection is lost. When the model is at the `lost` state, the Sender repeats trying to reconnect to the server. If the reconnection fails, the state goes back to `lost` by `nextIf`. When the model is at the `connected` state, the model randomly goes to the `stop` state. In this transition, the Sender starts to prepare for disconnection. Concretely, the transition weights of its Forwarder are set to be stable, and the transition `enable` is invoked. After the forwarder is set to be stable, the `end` transition is executed and the stored publish tokens are processed with `waitForCompletion` method in Paho. This method blocks until the message delivery corresponding the token finishes, or the designated timeout occurs. After processing all the stored tokens, the Sender disconnects from the server, and terminates the Forwarder by invoking the `end` transition in it.

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. An MQTT server stores only one message for each topic, therefore every message in this experiment must have a distinct topic. Therefore the topic of i-th message from the Sender is set to `randomSeed/i`.

Receiver

Figure 5.4 shows the state machine of the Receiver model. Similarly to the model of Sender, the Receiver model creates and launches an instance of Forwarder. Here the Forwarders of the Sender and the Receiver wait for TCP connection at different ports. In addition, the Receiver subscribes a topic filter. Since the messages from the Sender have topic of the form `randomSeed/i` and the Receiver need to subscribe messages with this form of topic for all i, the Receiver subscribes a topic filter `randomSeed/+`, where `+` means a one-level wildcard for topics.

In order not to miss the messages after the Receiver disconnects from the server, the `stop` transition can be executed only after the Sender finishes disconnection completely. This timing is managed by the `require` method in Modbat. The remaining transitions works the same as those of the Sender, except for the Receiver neither publishes messages nor waits for completion of them.

The test oracle is embedded at the end of the `end` transition in the Receiver, as an `assert` statement. The actual condition of the `assert` statement varies depending on the QoS of the messages in the experiment as shown in Table 5.2. 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 failure if this condition is violated, or uncaught exceptions are raised.

Table 5.2: QoS of messages and expected condition about message arrival

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

5.1.3 Results

We performed 50 test cases for each experiment. Each test cases took about hundreds milliseconds.

Table 5.3 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.

Table 5.3: Results of the experiments testing Paho

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

We describe the timeout in detail. The timeout happened in the method `waitForCompletion` in Paho, called in `end` transition of the Sender. This timeout means that the message delivery corresponding to the token has not been completed. However, the timeout only happened with QoS 0 and an unstable Sender, therefore this seems to 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 5.2 strictly. Namely, $P > R$ and $P < R$ held in some test cases with messages in QoS 0 and QoS 1, respectively. This result implies message loss actually occurred in the experiments in QoS 0, and message re-delivery actually performed in the experiments in QoS 1.

5.2 Test of Smart House

5.2.1 Method

We designed a model of a smart house, which runs thermometers and air conditioners in order to keep the room temperature. This system runs in unstable environments and in real time, therefore our extensions are useful to modelize the system.

The house has many devices communicating with MQTT clients, and runs an MQTT broker to relay messages from the devices to other devices. The house has 4x4 rooms and each room has a thermometer and an air conditioner. Each of a thermometer and an air conditioner runs an MQTT client in it, and connects to the server with the client. In addition, the house has a device named controller. The controller also runs an MQTT client to connect to the server and manages the room temperatures. The controller receives room temperatures from the thermometers and publishes messages to the air conditioners to control the room temperatures around the set temperature. Users of the house set temperature with the controller. In these experiments, the software running on the controller, thermometers and air conditioners are regarded as SUTs. It is expected that they work to keep the room temperatures as close to the set temperature as possible.

Each room in the house is identified by x and y coordinates. The temperature of each room is affected by its adjacent rooms and the mode of its air conditioner. The outside temperature also changes along with the time, which affects the temperature of the rooms facing the outer walls.

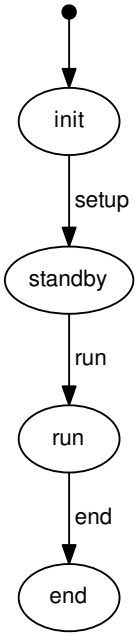

Figure 5.5: The top level model of a smart house

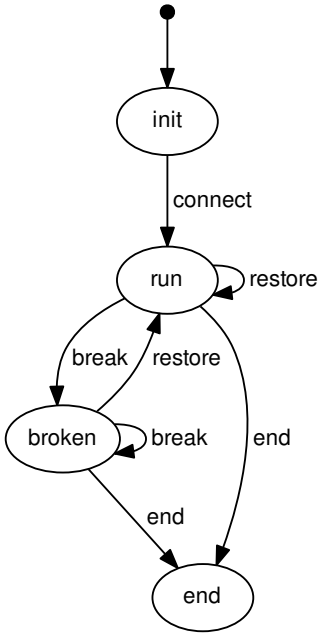

Figure 5.6: A model of Thermometer

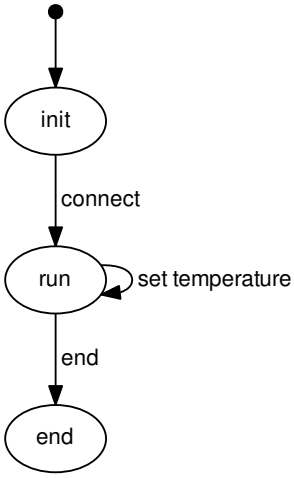

Figure 5.7: A model of Controller

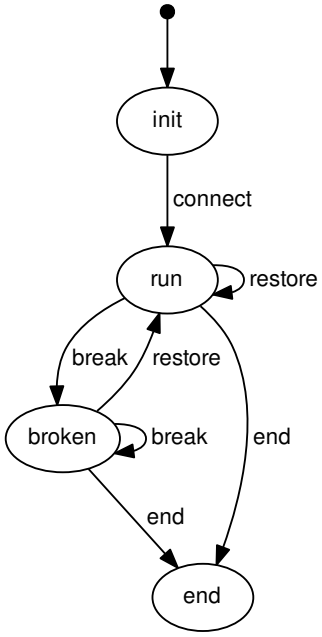

Figure 5.8: A model of Air Conditioner

In these experiments, the change of room temperatures are simulated along with the time change while device errors happen, and the effect of device errors are evaluated and visualized by comparing two houses in the same environment and with the different robustness of the devices.

These experiments also uses Mosquitto as the MQTT server, and the clients are implemented with the class `MqttAsyncClient` in Paho Java APIs.

5.2.2 Models

A model of the smart house consists of five types of model instances.

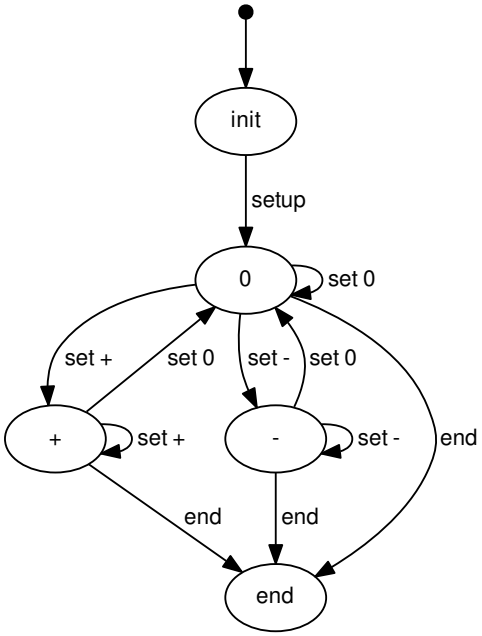

Figure 5.9: A model of Temperature Manager

Top Level Model

Figure 5.5 shows the state machine of the top level model of this experiment for testing the smart house. The instance of this model launches other model instances in the `setup` transition. The simulation runs while this model is in the `run` state. The transition `run` is set to stay for 3000 ms with `stay` method so that the simulation continues running for 3000 ms. When this model goes to the `end` state, other models start to terminate the simulation.

The instance of this model also stores the actual room temperatures, the actual modes of air conditioners and the outside temperature. At the beginning of a test case, the room temperatures and the outside temperature are set to 20.0 °C.

Thermometers

A thermometer is installed in each room in the smart house. Figure 5.6 shows the state machine of the model of a thermometer. When the transition labeled `connect` is executed, the MQTT client in the thermometer connects to the server. Then the thermometer spawns a thread, which repeats measuring the room temperature and publishing it to the server at the regular interval. This measured temperature may include error, and the intensity of the error may change depending on the state of the model, `run` or `broken`. In our experiments, the error is set to ±0.5 °C if the thermometer is at the `run` state, and set to ±10.0 °C if it is at the `broken` state. The transition between `run` and `broken` states has `stay` attribute with random time, which prevents from changing the state too often.

The messages are published in QoS 0, without waiting for completion of the delivery. Each message has the same topic, identified by the random seed of the test case and coordinates of the room.

Controller

The smart house has a controller to manage the overall system in the house. The controller has a set temperature, which is changed by the user in the house. The controller receives the temperatures from the thermometers and stores the temperatures for each room. This stored temperatures may be different from the actual temperatures. The stored temperatures are initially set to 20.0 °C, which are the same as the actual temperatures. The controller repeats operating the air conditioners to keep the room temperatures around the set temperature by publishing messages.

Figure 5.7 shows the state machine of the model of a controller. The model instance connects to the server in the transition `setup`. It also subscribes topics for receiving messages from the thermometers. At the same time, it spawns a thread that repeats publishing messages to the air conditioners at a regular interval. The actual algorithm for controlling the air conditioners varies depending on the experiments. The messages are published in QoS 2 and with a topic identified by the random seed and coordinates of the room.

The set temperature is set to 20.0 °C at the beginning of a test case. The transition labeled `set temperature` updates the set temperature by +1.0 °C, −1.0 °C or ±0.0 °C randomly. This transition has `stay` attribute with random time.

After the top level model goes to the `end` state, the controller can execute the `end` transition, which terminates its MQTT client.

Air Conditioners

An air conditioner in this system has three modes, stop, heating and cooling. We represent these modes as `0`, `1` and `-1` respectively.

Figure 5.8 shows the state machine of the model of an air conditioner. The transitions in this state machine have similar roles as those of a thermometer. We describe the difference from a thermometer in this section.

In the `connect` transition, the air conditioner also subscribes a topic to receive messages from the controller in QoS 2. A message from the controller specifies a mode by `0`, `1` or `-1`, and the air conditioner sets its mode according to the message in a callback function. Here, it sets its mode correctly if it is at the `run` state, but it sets its mode randomly if it is at the `broken` state.

Temperature Manager

The top level model launches a model instance of temperature manager, which is a model representing the environment of the smart house. The temperature manager is not a device in the house but calculates the outside temperature and room temperature in the simulation.

Figure 5.9 shows the state machine of the model of a temperature manager. In the `setup` transition, it spawns a thread to repeat calculating and updating the temperatures at regular interval. The states `0`, `+` and `-` represent the change of the outside temperature. The outside temperature increases or decreases while the model is in the `+` or `-` state, respectively.

The room temperatures are calculated in a discretized time interval. Let x be a room, $T_x(t)$ be the temperature of room x at time t and A be a set of rooms adjacent to x. Here we regard the outside as an adjacent room to x if x faces the outer wall of the house. And let $c_{x,y}$ be the conduction coefficient between room x and room y, r be the power of air conditioners, and $ac_x(t)$ be the mode

of the air conditioner in room x at time t. Then the room temperature at next time step is calculated by the formula 5.1.

$$T_x(t + \Delta t) = T_x(t) + \Delta t \left(\sum_{y \in A} c_{x,y} \left(T_y(t) - T_x(t) \right) + r \cdot ac_x(t) \right) \tag{5.1}$$

This formula intuitively means that the rate of change of the room temperature is proportional to the temperature difference from the adjacent rooms. In these experiments, $c_{x,y}$ is set to $0.0001\,^\circ\mathrm{C/ms}$ if either x or y is the outside, otherwise set to $0.0002\,^\circ\mathrm{C/ms}$.

5.2.3 Evaluation of the Effects

In order to evaluate the difference among algorithms or the effect of the errors in the experiments, two instances of the house is executed in parallel and their outputs are compared. The two instances have the same environment, including the model states, the outside temperature and the set temperature. However, they may have different MQTT clients in the devices, control algorithms, the modes of the air conditioners and devices with different robustness, which result in producing different room temperatures. Then we can evaluate the performance of the two algorithms or the effect of device errors, by comparing the two outputs.

5.2.4 Test Variations

We conducted running simulation with four pairs of different settings. We distinguish the instances in a pair by "System A" and "System B".

Experiment 1: Effect of Thermometer Errors

This experiment evaluates the effect of thermometer errors. In this experiment, all the air conditioners always work correctly. In System A, all the thermometers work correctly even when its model state is at the `broken` state. In System B, the thermometer publishes a temperature with large error when it is at the `broken` state.

Experiment 2: Effect of Air Conditioner Errors

This experiment evaluates the effect of the errors of the air conditioners. All the thermometers work correctly in this experiment. In System A, all the air conditioner always work correctly. Namely, an air conditioner sets its mode correctly as designated by the controller. In System B, the air conditioner may be broken, so it sets its mode randomly when it is at the `broken` state.

Experiment 3: Control Algorithm Comparing to Previous Temperatures

This experiment evaluates the performance of an algorithm of the controller to detect the errors of the thermometers. Both thermometers and air conditioners may be broken as explained in Section 5.6 and Section 5.8. System A has a controller with a naive algorithm, which turns on cooling when the room temperature is higher than the set temperature and vice versa. The controller always "believes" the temperatures from the thermometers. The algorithm of the controller in System B differs in that when it receives a temperature but the difference

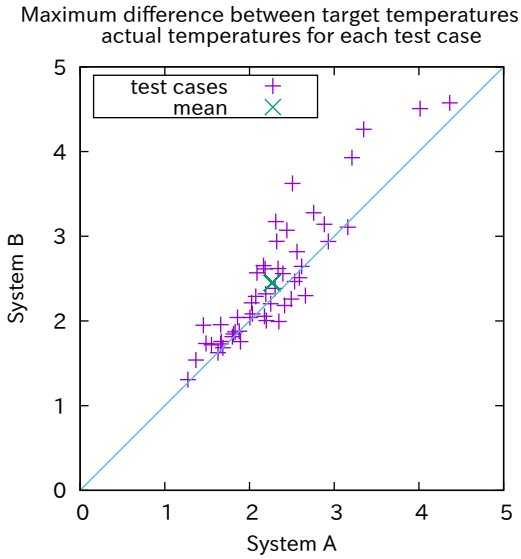

Figure 5.10: The effect of thermometer errors

between the temperature and the stored temperature of the room is larger than a certain threshold, it discards the new temperature and keeps using the stored temperature. The threshold is set to 2.0 °C in this experiment.

Experiment 4: Control Algorithm Comparing to Adjacent Room Temperatures

This experiment evaluates another algorithm of the controller. Thermometers, air conditioners and the controller in System A works as the same as those of the previous experiment. The controller in System B always stores the new temperatures from the thermometers, but the controller checks the room temperatures when it operates the air conditioners. It compares the room temperature and the mean of the temperatures in the adjacent rooms, and if the difference is larger than a threshold, it uses the mean instead of the stored temperature to control the air conditioner. The threshold is set to 2.0 °C in this experiment.

5.2.5 Results

We performed 50 test cases for each experiment and plotted the maximum difference between the room temperatures and the set temperature across the time and all the rooms, comparing the paired models. In these figures, a point of the shape + corresponds to a test case, and the point shows that System A was better than System B if it is over the diagonal and vice versa.

Figure 5.10 shows the effect of thermometer errors and Figure 5.11 shows the effect of air conditioner errors. These two figures show that both air conditioner errors and thermometer errors enlarge the temperature errors.

Figure 5.12 and Figure 5.13 show the effectiveness of two controller algorithms. In these experiments, both algorithms improved the temperature errors. However, the figures also show that the difference among test cases had major impact on the temperature errors and there were some test cases the algorithms made the temperature errors worse.

In addition, we visualized the change of the room temperatures as gif animated heat maps. Figure 5.14 shows the heat map at a certain moment in a test case in the Experiment 1. The top two heat maps show the temperature and the

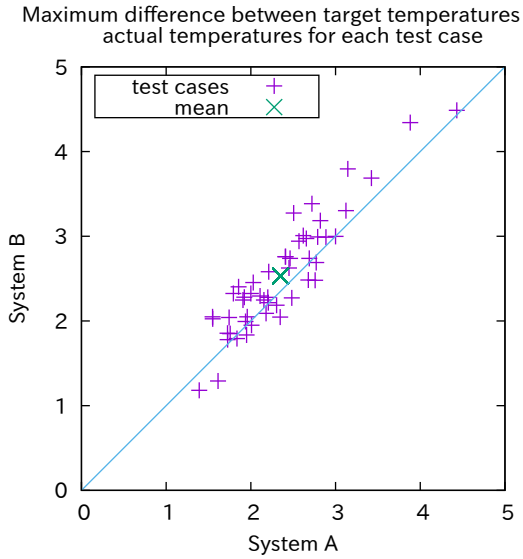

Figure 5.11: The effect of air conditioner errors

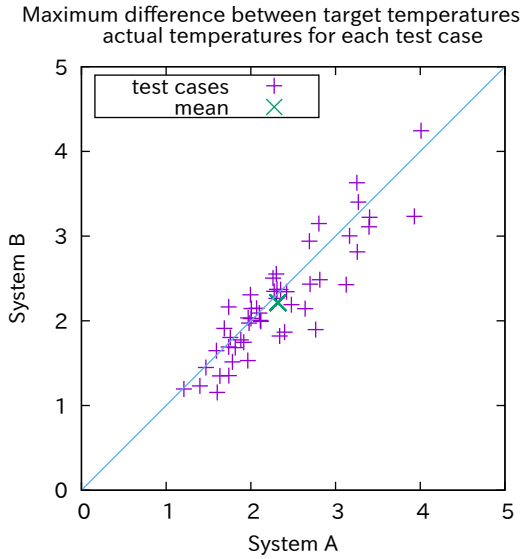

Figure 5.12: Comparison between a naive algorithm and an algorithm comparing to previous temperature

mode of the air conditioner in each room in System A and System B. Each cell corresponds to a room and the two figures in the cell represent the temperature and the mode of the air conditioner respectively. The left bottom heat map shows the difference of the temperatures in each room between System A and System B. If the value is positive, it shows that the room temperature in System B is higher than that in System A. The right bottom heat map shows the difference of the temperature error in each room between System A and System B. Namely, the values in this heat map is expressed as $|T_B - S| - |T_A - S|$, where T_A and T_B are the room temperature in System A and System B respectively, and S is the set temperature at the moment. If the value is positive, it shows that the room in System B has the closer temperature to the set temperature than System A.

These visualizations clearly show the difference between two models.

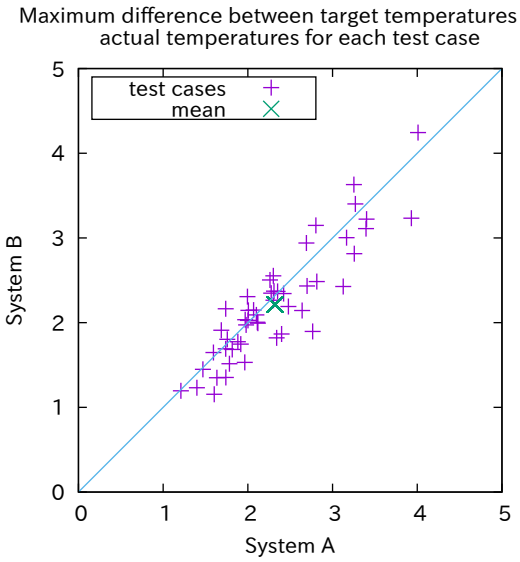

Figure 5.13: Comparison between a naive algorithm and an algorithm comparing to adjacent temperatures

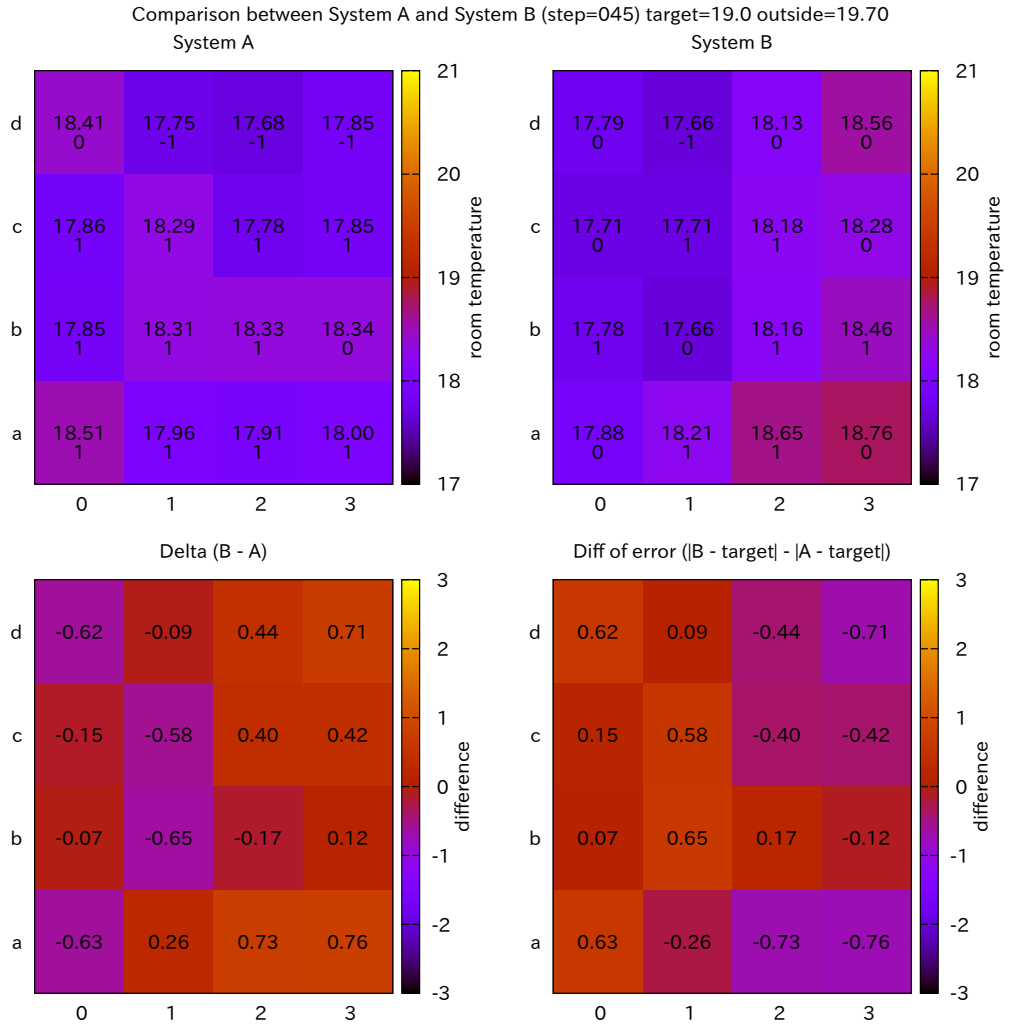

Figure 5.14: A heat map comparing System A (without errors) and System B (with thermometer errors)

Chapter 6

Conclusions

In this study, we have defined extensions for performing model-based tests suitable for IoT environments and implemented them in Modbat.

The first extension, timed extended finite state machines, enable modeling systems running in real time. The second extension, dynamic weight change, enables the transition probabilities of EFSMs dynamically in test executions. This extension realizes modeling devices whose error rates may change. The third extension, transition invocation, enables invoking transition functions forcibly by other transition functions or callback functions. This extension realizes modeling devices which handle errors in a simple and natural way. We have shown that these extensions are effective to describe models of IoT systems briefly with high expressiveness.

We have also proposed packet forwarders, in order to simulate unstable TCP network environments during test execution by software without modifying the SUTs. By using packet forwarders, network disconnection and network delay can be controlled by transition functions in test models. We have also shown that packet forwarders actually simulate network connection loss through testing MQTT implementations.

Chapter 7

Future Work

The timed extended finite state machines in our definition have limited expressiveness compared to regular timed automata [3]. Further extensions are desired for more expressiveness with a simple syntax.

The packet forwarder in this thesis relays TCP packets but it may be applied to other transport layer protocols such as UDP. Since the order of the packets is not guaranteed in UDP, the UDP forwarder will have more complicated structure and operations.

The packet forwarder simulates network errors by controlling TCP layer. Though it simulates errors for applications using APIs of TCP, the responses from the APIs may be different from the cases that actual network errors occur. Therefore it is necessary to evaluate the difference and to develop methods to simulate network errors in ways closer to the actual errors.

In the experiments in Section 5.2, the models of devices break down just probabilistically. However, the actual devices may be broken because of various reasons, therefore the error rate may vary depending on the environment. In order to create test models breaking like actual devices, it is necessary to construct a kind of probabilistic models such as Markov models.

It is useful to reproduce errors found in tests for finding the causes of the errors. However, in IoT systems, since multiple devices communicate and their software may run multiple threads, their behavior may be non-deterministic. In other words, the system may produce different output even if the test tool executes In order to reproduce outputs from communicating systems, net-iocache may be used to reproduce network communication [6]. However, net-iocache is a method for software model checking, therefore there may be difficulty for applying net-iocache to model-based testing.

References

[1] Eclipse Paho - MQTT and MQTT-SN software. `http://www.eclipse.org/paho/`.

[2] Mosquitto. `https://mosquitto.org/`.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[4] Cyrille Artho and Armin Biere. Modbat. `https://people.kth.se/~artho/modbat/`, 2016.

[5] Cyrille Artho, Quentin Gros, Guillaume Rousset, Kazuaki Banzai, Lei Ma, Takashi Kitamura, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto. Model-based API testing of Apache ZooKeeper. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 288–298. IEEE Computer Society, 2017.

[6] Cyrille Artho, Watcharin Leungwattanakit, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto. Cache-based model checking of networked applications: From linear to branching time. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 447–458. IEEE Computer Society, 2009.

[7] Cyrille Valentin Artho, Armin Biere, Masami Hagiya, Eric Platon, Martina Seidl, Yoshinori Tanabe, and Mitsuharu Yamamoto. Modbat: A model-based API tester for event-driven systems. In Valeria Bertacco and Axel Legay, editors, *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, volume 8244 of *Lecture Notes in Computer Science*, pages 112–128. Springer, 2013.

[8] Andrew Banks and Rahul Gupta. MQTT version 3.1.1. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`, 2014.

[9] Carsten Bormann. CoAP — constrained application protocol — overview. `http://coap.technology/`, 2014–2016.

[10] Kwang-Ting Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In Alfred E. Dunlop, editor, *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993.*, pages 86–91. ACM Press, 1993.

[11] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors,

Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000., pages 268–279. ACM, 2000.

[12] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An introduction to the testing and test control notation (TTCN-3). *Computer Networks*, 42(3):375–403, 2003.

[13] Seungjae Han, Kang G Shin, and Harold A Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 204–213. IEEE, 1995.

[14] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[15] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Trans. Computers*, 44(2):248–260, 1995.

[16] Lausanne (EPFL) Lausanne. The Scala programming language. `https://www.scala-lang.org/`, 2002–2018.

[17] Shinho Lee, Hyeonwoo Kim, Dong-kweon Hong, and Hongtaek Ju. Correlation analysis of MQTT loss and delay according to QoS level. In *The International Conference on Information Networking 2013, ICOIN 2013, Bangkok, Thailand, January 28-30, 2013*, pages 714–717. IEEE Computer Society, 2013.

[18] Kristiyan Mladenov, Stijn van Winsen, Chris Mavrakis, and KPMG Cyber. Formal verification of the implementation of the MQTT protocol in IoT devices. 2017.

[19] Rickard Nilsson. ScalaCheck. `https://www.scalacheck.org/`, 2015.

[20] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2008.

[21] Zary Segall, Dalibor F. Vrsalovic, Daniel P. Siewiorek, David A. Yaskin, J. Kownacki, James H. Barton, R. Dancey, A. Robinson, and T. Lin. FIAT-fault injection based automated testing environment. In *Proceedings of the Eighteenth International Symposium on Fault-Tolerant Computing, FTCS 1988, Tokyo, Japan, 27-30 June, 1988*, pages 102–107. IEEE Computer Society, 1988.

[22] David T. Stott, Gregory L. Ries, Mei-Chen Hsueh, and Ravishankar K. Iyer. Dependability analysis of a high-speed network using software-implemented fault injection and simulated fault injection. *IEEE Trans. Computers*, 47(1):108–119, 1998.

[23] Rickard Svenningsson, Jonny Vinter, Henrik Eriksson, and Martin Törngren. MODIFI: A model-implemented fault injection tool. In Erwin Schoitsch, editor, *Computer Safety, Reliability, and Security, 29th International Conference, SAFECOMP 2010, Vienna, Austria, September 14-17, 2010. Proceedings*, volume 6351 of *Lecture Notes in Computer Science*, pages 210–222. Springer, 2010.

[24] Dinesh Thangavel, Xiaoping Ma, Alvin C. Valera, Hwee-Xian Tan, and Colin Keng-Yan Tan. Performance evaluation of MQTT and coap via a common middleware. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, April 21-24, 2014*, pages 1–6. IEEE, 2014.

[25] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test., Verif. Reliab.*, 22(5):297–312, 2012.