Model-Based Testing IoT Communication via Active Automata Learning

Martin Tappler Bernhard K. Aichernig
Institute of Software Technology
Graz University of Technology, Austria
{martin.tappler, aichernig}@ist.tugraz.at

Roderick Bloem
Institute of Applied Information Processing and Communications
Graz University of Technology, Austria
roderick.bloem@iaik.tugraz.at

***Abstract*—This paper presents a learning-based approach to detecting failures in reactive systems. The technique is based on inferring models of multiple implementations of a common specification which are pair-wise cross-checked for equivalence. Any counterexample to equivalence is flagged as suspicious and has to be analysed manually. Hence, it is possible to find possible failures in a semi-automatic way without prior modelling.**

We show that the approach is effective by means of a case study. For this case study, we carried out experiments in which we learned models of five implementations of MQTT brokers/servers, a protocol used in the Internet of Things. Examining these models, we found several violations of the MQTT specification. All but one of the considered implementations showed faulty behaviour. In the analysis, we discuss effectiveness and also issues we faced.

I. Introduction

Active automata learning has gained increasing attention of the verification and testing community in recent years. There exist several different approaches to this kind of learning. Many of them are based on or related to the L^* algorithm by Angluin [9]. As such these approaches share a strong connection to conformance testing [10]. In both areas, learning and conformance testing, the goal is to gain knowledge about the behaviour of a black-box system, by executing tests/queries[1] and analysing corresponding observations. However, in the former we are interested in the synthesis aspect, i.e. we want to infer a model, whereas in the latter, we perform an analysis task, i.e. we check conformance to a given model.

This opens up the possibility to combine these approaches. Aarts et al. [4] have for instance shown how to combine learning, testing and verification. They learned the model of a reference implementation of the bounded retransmission protocol and checked equivalence between this model and several mutated (faulty) implementations via two different techniques. (1) They performed model-based testing of the mutants using the learned reference model. (2) Additionally, they also learned models of the mutants and subsequently checked equivalence between the inferred models of the specification and of each of the mutants. In this paper, we will follow an approach similar to the latter. Both approaches differ most significantly in the kind of implementations considered. They actually generated Java applications from models with a known structure.

[1]Tests are often called (membership/output) queries in active automata learning.

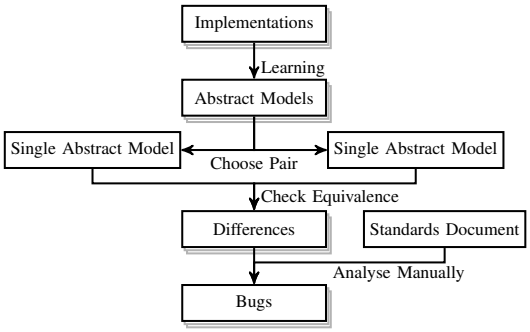


Fig. 1. Overview of bug-detection process.

Furthermore, the faulty implementations have been created artificially by seeding known errors into the reference model. On the contrary, we do not know anything about the structure of the analysed implementations. We merely know that they implement a common specification given in natural language.

In order to detect faults in the considered implementations, we thus propose the following learning-based approach. In a first phase, we learn (Mealy-machine) models of several different implementations of some standardised protocol or operation. These models are then pair-wise cross-checked with respect to some conformance relation. All counterexamples to conformance are then analysed manually by consulting a given standards document. This may either reveal a bug in one or both implementations corresponding to the counterexample, or it may reveal an underspecification of some aspect of the standard. The process we follow is also depicted in Fig. 1.

This approach apparently cannot detect all possible faults because specific faults may be implemented by all examined implementations. In addition to that, the fault-detection capabilities are limited by the level of abstraction used for learning. This gives rise to two research questions we aim at answering in the following. Is it possible to effectively detect non-trivial faults using our approach despite the necessary severe abstraction? What are the limitations of our approach? Related to the second question, we will discuss opportunities for future research in order to mitigate the identified limitations.

For this purpose we will analyse the behaviour of five different brokers implementing Message Queuing Telemetry Transport (MQTT) version 3.1.1 [14], a protocol standardised by the

978-1-5090-6031-3/17 $31.00 © 2017 IEEE
DOI 10.1109/ICST.2017.32

International Organization for Standardization (ISO) [32]. The MQTT protocol is a lightweight publish/subscribe protocol and therefore well-suited for resource-constrained environments such as the Internet of Things (IoT). This is the main reason we have chosen MQTT for our case study as we want to investigate verification techniques in the context of the IoT. Furthermore, we consider broker implementations since they constitute central communication units, hence it is essential to assure their correct operation for a reliable communication in the IoT. Basically, brokers allow clients to subscribe and to publish to topics. If a message is published, the broker forwards it to all clients with matching subscriptions.

The main contribution of this paper is thus the presentation and empirical evaluation of the mentioned approach based on learning experiments with five different black-box systems. More concretely, we learned models of five different MQTT brokers, systems used in IoT communication. In our analysis, we will discuss failure-detection with a focus on required effort, issues related to runtime, and general challenges.

We are not aware of any case study applying conformance checking between learned models in a purely black-box setting. Neither do we know of any case study focusing on the verification of implementations of the IoT protocol MQTT.

The structure of this paper is as follows. In Section II, we will discuss related work in the area of testing and verification. In Section III, we will introduce the used modelling formalism and active automata learning. Section IV introduces the approach we follow in our case study. We will present implementation details and results obtained from learning experiments in Section V. We conclude in Section VI with a summary of our findings and a discussion of future work.

II. Related Work

In this section, we will review related work with a focus on the combination of active automata learning and verification. We already mentioned the work by Aarts et al. [4] in the introduction. Although they also used Mealy machines and a similar technique to distinguish mutated (faulty) implementations from a reference implementation, our setting is different. While Aarts et al. consider learning of more complex models in terms of states and number of inputs, we deal with different challenges. The challenges we face are mainly related to the network communication. We have to deal with long response times and with completely unknown system structure. Basically, we do not know whether our implementations behave like Mealy machines, a property fulfilled by their implementations generated from a known UPPAAL model. Additionally, they created the mutated implementations by seeding faults into the reference implementation, thus they also know beforehand that the checked systems share a similar structure.

We apply both testing and exhaustive equivalence checking in our case study. The former is used during learning and the latter during the analysis of learned models. Early work in this area has been performed by Peled et al. [40] and Groce et al. [26]. They combine testing and model checking, in order to analyse black-box systems as well. Additionally, they also apply active automata learning to infer the structure of the analysed systems from the observations made during testing.

We make use of the assumption that a learned model faithfully represents the corresponding black-box system. Hence, we can simulate tests of a system by executing tests on the learned automaton. This is possible since we already tested for equivalence/conformance between the black-box system and the hypothesis automaton during learning. Berg et al. [10] discuss the correspondence between conformance testing and learning in more detail and show similarities and differences.

In the analysis of the transport layer security (TLS) protocol, de Ruiter and Rutten consider a similar learning setup [18]. They investigate the behaviour of several implementations of the TLS protocol via active automata learning, but they manually analyse the inferred models. While we are interested in any error that can be found on state-machine level, they specifically target security-related flaws. Beurdouche et al. [12] also targeted state-machine based flaws. They followed a test-based approach, but generated tests from a known model via some heuristics. Thereby they checked for specific faults, like faults related to skipping mandatory steps in a protocol.

Fiterău-Broştean et al. also performed case studies involving active learning of models of several implementations of a single protocol [20], [21]. More concretely, they learned Mealy-machine models of transmission control protocol (TCP) implementations. They additionally applied model-checking [21] in order to verify properties of the composition of client and server implementations.

III. Preliminaries

A. Mealy Machines

We will use Mealy machines as modelling formalism because they are well-suited to model reactive systems and they have successfully been used in contexts combining learning and some form of verification [18], [21], [36]. In addition to that, the application of Mealy machines allows us to use the existing Java-library LearnLib [31] which provides efficient algorithms for learning Mealy machines.

Basically, Mealy machines are finite state automata with inputs and outputs. The execution of such a Mealy machine starts in an initial state and by executing inputs it changes its state. In addition to that, exactly one output is produced in response to each input. We will refer to Mealy machines also as state machines in the remainder of this paper. Formally, Mealy machines can be defined as follows.

Definition III.1 (Mealy Machines)**.** A Mealy machine is a 6-tuple $\langle Q, q_0, I, O, \delta, \lambda \rangle$ where

- Q is a finite set of states,
- q_0 is the initial state,
- I/O is a finite set of input/outputs symbols,
- $\delta : Q \times I \to Q$ is the state transition function, and
- $\lambda : Q \times I \to O$ is the output function

We require Mealy machines to be input enabled and deterministic. The former demands that an output and a successor

state must be defined for all inputs and all states, i.e. δ and λ must be surjective. A Mealy machine is deterministic if it defines at most one output and successor state for every pair of input and state, thus δ and λ must be functions in the mathematical sense.

We will now introduce some notational conventions for sequences of input/output symbols $s \in S^*$, where $S = I$ or $S = O$. Let $s' \in S^*$ be another sequence, then $s \cdot s'$ denotes the concatenation of these sequences. The empty sequence is represented by ϵ. We implicitly lift single elements to sequences, thus for $e \in S$ we have $e \in S^*$. As a result, the concatenation $s \cdot e$ is also defined.

Furthermore, δ and λ are extended to sequences of inputs in the standard way. Let $s \in I^*$ be a sequence of inputs and $q \in Q$ be a state of a Mealy machine, then $\delta(q, s) = q' \in Q$ is the state reached by executing s starting in state q. Given a sequence of inputs $s \in I^*$ and a state $q \in Q$, the output function $\lambda(q, s) = t \in O^*$ returns the outputs produced in response to s executed in state q.

Finally we need a basis for determining whether two Mealy machines are equivalent. Equivalence is usually defined with respect to outputs [2], i.e. two deterministic Mealy machines are equivalent if they produce the same outputs for all input sequences. We say that a Mealy machine $\langle Q_1, q_{0_1}, I, O, \delta_1, \lambda_1 \rangle$ is equivalent to another Mealy machine $\langle Q_2, q_{0_2}, I, O, \delta_2, \lambda_2 \rangle$ iff $\forall s \in I^* : \lambda_1(q_{0_1}, s) = \lambda_2(q_{0_2}, s)$. A counterexample to equivalence is thus an $s \in I^*$ such that $\lambda_1(q_{0_1}, s) \neq \lambda_2(q_{0_2}, s)$.

B. Active Automata Learning

In the following, we will consider learning algorithms operating in the minimally adequate teacher (MAT) framework proposed by Angluin [9]. These algorithms infer models of black-box systems, also referred to as systems under learning (SULs), through interaction with a so-called teacher.

1) Minimally Adequate Teacher Framework: The interaction is carried out via two types of queries posed by the learning algorithm and answered by a minimally adequate teacher. These two types of queries are usually called *membership queries* and *equivalence queries*. In order to understand these basic notions of queries consider that Angluin's original L^* algorithm was used to learn a deterministic finite automaton (DFA) representing a regular language known to the teacher [9]. Given some alphabet, the L^* algorithm repeatedly selects strings and asks membership queries in order to check whether these strings are in the language to be learned. The teacher may answer either *yes* or *no*.

After some queries the learning algorithm uses the knowledge gained so far and forms a hypothesis, i.e. a DFA consistent with the obtained information which should represent the regular language under consideration. The algorithm presents the hypothesis to the teacher and issues an equivalence query in order to check whether the language to be learned is equivalent to the language represented by the hypothesis automaton. The response to this kind of query is either *yes* signalling that the correct DFA has been learned or a counterexample to equivalence. Such a counterexample is a witness showing that the learned model is not yet correct, i.e. it is a word from the symmetric difference of the language under learning and the language accepted by the hypothesis.

If a counterexample is provided then learning algorithms incorporate this counterexample into their data structures and start a new *round* of learning. The new round again involves membership queries and a concluding equivalence query.

This general mode of operation is used by basically all algorithms in the MAT framework with some adaptations. These adaptations may for instance enable the learning of Mealy machines which we will describe in the following.

2) Learning Mealy Machines: Margaria et al. [36] and Niese [39] were one of the first to infer Mealy-machine models of reactive systems by applying a learning algorithm based on L^*. Another learning algorithm for Mealy machines, also based on L^*, has been presented by Shahbaz and Groz [41]. They basically reuse the structure of L^*, but instead of membership queries, they pose *output queries*. Thus, instead of checking whether a string is in some language, they provide an input string to the teacher and the teacher responds with the corresponding output string.

For a more practical discussion of learning consider the instantiation of a teacher. Usually we want to learn the behaviour of a black-box SUL of which we only know the input and output interface. Hence, output queries are conceptually simple: inputs are provided to the SUL and it produces some outputs. However, there is a slight difficulty hidden. Like Angluin [9], Shahbaz and Groz [41] assume that output queries provide outputs in response to inputs executed from the **initial** state. Consequently, we need to have some means to reset a system. Since we are dealing with a black-box system, we normally cannot check for equivalence with a hypothesis. In practice, it is thus necessary for a teacher used in a learning algorithm to approximate equivalence queries somehow. This can for instance be achieved via conformance testing as implemented in LearnLib [31].

To summarise, a learning algorithm for Mealy machines relies on three operations:

reset: resets the SUL

output query: performs a single test executing a sequence of inputs and recording the outputs

equivalence query: performs a set of tests comparing the outputs of the SUL and the current hypothesis

Hence, the teacher is usually some component interacting with the SUL in order to implement these operations. An equivalence query results in a positive answer if all tests pass, i.e. the SUL produces the same outputs as the hypothesis automaton. If there is a test for which their outputs differ, the corresponding sequence of inputs is presented to the learning algorithm as a counterexample. The interaction between SUL, teacher and learning algorithm is also depicted in Fig. 2.

Note that due to the incompleteness of testing the learned model may be incorrect if the equivalence query is replaced with conformance testing. If for instance the W-method by Vasilevskii [45] and Chow [16] is used to derive a confor-

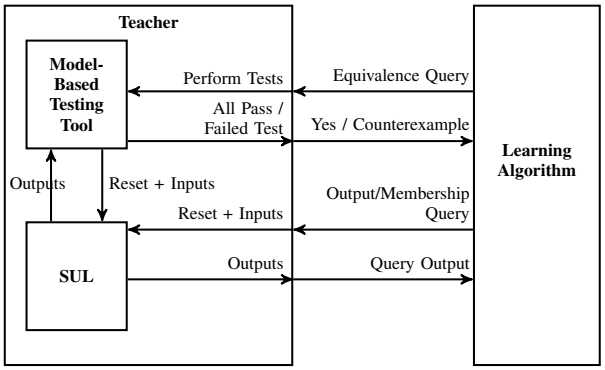


Fig. 2. The interaction between SUL, teacher and learning algorithm (based on a figure by Smeenk et al. [42]).

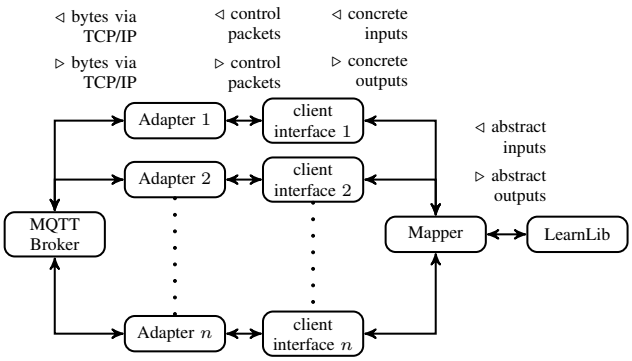


Fig. 3. The architecture we used for learning of Mealy-machine models of MQTT brokers.

mance test suite, the learned model may not be correct if assumptions placed on the maximum number of states of the SUL do not hold.

IV. Approach

In this section, we will discuss our approach in more detail. We will start with a discussion of learning Mealy-machine models in which we also highlight some technical considerations to enable the task of learning such models.

A. Learning

1) Architecture: Two aspects influence the architecture of a learning environment for MQTT. Firstly, we need to account for dependencies between clients. Unlike in pure client/server-settings, like in the TLS protocol [18] or the TCP [3], [21], it is not sufficient to simulate one client to adequately infer a model of the server/broker in MQTT. We need to control multiple clients and record the messages each one has received.

Secondly, we need to cope with the enormous amount of possible inputs, i.e. the large number of packets we can send to the brokers. This, however, is a general problem of active automata learning in the MAT framework and an issue for learning almost all non-trivial systems. To deal with this problem, we introduce a mapper component performing abstraction and concretisation [2], [3], [27]. However, unlike in the cited work, we do not refine our abstractions in an iterative manner, but rather use a static mapper throughout the learning phase. If non-determinism arising from abstraction or from the processing of outputs in general was detected, we manually adapted the learning setup in an appropriate way.

Fig. 3 shows the architecture of our learning setup. The SUL, an MQTT broker, is shown on the left-hand side. In order to learn its behaviour we control its environment made up of several clients which basically consist of the adapter blocks and the client-interface blocks. The adapter handles communication-related tasks, whereas the client-interface components implement a client library with a simple interface and default values for control-packet parameters.

The right-hand side of Fig. 3 shows the components responsible for learning. LearnLib implements several algorithms for actively learning Mealy machines. During the execution of such an algorithm, LearnLib interacts with a mapper component by choosing and executing one of the available abstract inputs. The mapper concretises abstract inputs by choosing one of the clients and executing some action with concrete values. Note that the sending of packets and serialisation tasks performed during an action are handled by the clients. As soon as outputs are available, the mapper collects them from all clients and abstracts them, creating one abstract output symbol in response to each abstract input.

2) Practical Considerations:

a) Timeouts: We already noted that there are delays between the transmission of messages to the broker and the receipt of corresponding responses. These are inevitably present since network communication is involved, which is actually asynchronous. In addition to that a system may not send any message, i.e. not produce any output at all. A more faithful model of such a system would thus for instance be a timed automaton [8].

However, instead of changing the modelling formalism we followed a more pragmatic approach. We basically set a configurable timeout for the receipt of messages like de Ruiter and Poll [18]. All messages received before reaching this timeout are processed by the mapper component to form an abstract output symbol. If we do not receive any message, we say that the system is quiescent and the mapper produces the corresponding output symbol *Empty*. This is similar to the way the absence of outputs is handled in input-output conformance (**ioco**) testing except that there quiescent behaviour is represented by the symbol δ [43].

b) Processing Outputs: There are several steps involved in the processing of messages received from a broker. After deserialisation, relevant information is extracted from messages. Since a single client may receive multiple messages in response to a single input sent by one of the clients, these message groups have to be processed to create one output per client. We therefore sort the messages received by each client alphabetically to ensure determinism. This is necessary as messages may arrive in any order. Afterwards we concatenate the sorted messages. If a client does not receive any message,

we interpret this as having received a single message *Empty*. The outputs of individual clients are concatenated to create a single abstract output.

While some messages may arrive in any order, the MQTT specification also places restrictions on message ordering [14]. As a result, we lose relevant information by sorting. Since alternatively we would need to learn non-deterministic models, this way of handling outputs represents a tradeoff between completeness and efficiency.

c) Restrictions Placed on MQTT Functionality: We had to exclude some features of MQTT from our analysis as they cannot adequately be modelled using Mealy machines. The MQTT specification, e.g., includes a ping functionality which would require learning of time-dependent behaviour.

Additionally, there are dependencies between sent and received data, e.g., identifiers sent in acknowledgements should match identifiers of the acknowledged messages. For this reason, we send acknowledgements automatically from client to broker and do not learn behaviour related to that. This opens up a possibility for future work: it would be possible to carry out experiments with the tool Tomte [1], [2]. This tool is able to learn a restricted class of extended finite state machines (EFSMs), Mealy machines with guards and state updates. These EFSMs are expressive enough to model the acknowledgement mechanism.

It should be emphasised that the excluded features do not affect the parts of MQTT we learn.

B. Learning-Based Testing

We describe our approach to testing in the following. Roughly speaking, we test conformance between implementations and flag traces leading to conformance violations as suspicious. We do so by learning models of the concerned systems and subsequent equivalence checking of those models.

For a more in-depth discussion of the topic, assume that a model learned with the approach described above faithfully represents the SUL under the abstraction applied by the mapper. As a result, we can execute tests on the model and thereby implicitly perform tests on the SUL under the same abstraction. In other words, we can simulate a testing environment similar to Fig. 3, but with LearnLib replaced by a conformance testing component.

Note that this differs from the usual approach to model-based testing [44]. In general, a model is assumed to be given which can be used for generating tests and as a test oracle. The former is still possible with learned models, i.e. we can generate tests according to some criterion. The latter, however, is problematic as we cannot be sure whether the model is correct. A model learned by observing a faulty implementation will also be incorrect. To circumvent this problem, we use the learned model of another implementation as oracle.

We thus test whether some learned model conforms to another learned model and thereby we implicitly test conformance between the implementations. Since we do not know anything about the correctness of either implementation, we do not consider conformance violations to be errors, but rather flag traces leading to conformance violations as suspicious. After performing all tests of a conformance test suite, we manually investigate all traces which show non-conformance in order to determine whether any of the implementations violates the specification (i.e. the MQTT specification [14]).

This may reveal zero, one or two errors depending on whether the specification allows some freedom of implementation for the corresponding functionality, and whether one or both implementations implemented the functionality in a wrong way. Not all errors are detected for two reasons: (1) it may not be possible to detect some errors because of abstraction and (2) we do not detect an error if it is equally present in all implementations. The first problem is inherent to model-based testing, while the second problem is directly related to our approach. To overcome this issue and to decrease the probability of missing errors we compared the behaviour of five implementations instead of just two.

In order to cope with the large number of abstract inputs, we decided to create several sets of abstract input alphabets. In other words, we learned distinct models for subsets of the set of all abstract inputs. We thus had to implement as many mappers as input subalphabets. These subsets of inputs have been chosen in a way such that inputs within some subset have interesting dependencies. As we thereby also place implicit assumptions about independence relationships between inputs from different sets, the effectiveness of fault-detection may be impeded by this approach. Issues such as the effectiveness of the overall approach will be discussed in Section V. A similar approach has been followed by Smeenk et al. [42] for equivalence testing. In addition to the complete alphabet containing all inputs, they identified a subset of inputs relevant to initilialisation which they tested more thoroughly.

The separation into subsets of the complete input alphabet led to the following model-based testing process.
For each input alphabet:
1) Learn a model m_i of each implementation i
2) For each pair (m_i, m_j) of learned models:
    1) Check equivalence
    2) For each counterexample c to equivalence
        1) Test c on implementations i and j
        2) Analyse manually if outputs of i and j are correct

Note that if we find a counterexample to equivalence, i.e. a suspicious trace c, we test it on the corresponding implementations. We do so to ensure that c actually shows a difference between the implementations and is not the result from an error introduced by learning. Although active automata learning is in general sound, this may happen because we only approximate equivalence queries by conformance testing.

As we check conformance on model-level, we can also use techniques other than testing. We could, e.g., use external tools such as CADP to check equivalence [5], encode the problem as reachability problem and use satisfiability modulo theories (SMT) solvers for the task [6], or use a graph-based approach [7], [13]. We will actually use a graph-based approach, whereby we roughly interpret Mealy machines as labelled transition systems (LTSs) and build a synchronous

product with respect to a conformance relation [19], [49]. This will be described in more detail in Section V.

d) Comparison to Traditional Model-Based Testing: Now that we introduced the approach, we can discuss the effort required to perform learning-based testing in comparison to the effort required for traditional model-based testing. Some kind of adapter and abstraction component has to be implemented for both techniques, thus we address the effort related to interpretation of requirements for a comparison.

Usually in model-based testing, a large set of requirements stated in natural language has to be formalised which is both labour-intensive and error-prone. This, however, is not required in the learning-based approach. To perform the case study, we skimmed through the MQTT-specification to get a rough understanding of the protocol in general. Afterwards, we identified interesting interactions between control packets and specific parameters thereof to implement mappers. This can be compared to the definition of scenarios encoding test purposes, e.g , used by Spec Explorer [23]. In addition to that, we only had to analyse parts of the specification in more depth which correspond to suspicious traces.

Hence, less manual effort is required. It should be noted, though that this comes at the cost of decreased control of the testing process. While it is usually possible to direct the test case generation through test selection criteria [44], the tests performed for learning are selected by learning algorithms.

e) The Role of Learning: A question that comes to mind concerns the role of learning in our approach. Why do we actually learn Mealy machines? It would also be possible to generate some test suite, e.g. randomly, run the test suite on all implementations and check whether differences in outputs exist. However, learning offers two benefits. Firstly, it provides us with a model which can be used for further verification tasks such as model-checking [21]. In addition to that, it essentially defines a stopping criterion. Stated differently, we assume that we have tested adequately and can stop testing as soon as we can derive a correct system model. A similar approach is, e.g., followed by Meinke and Sindhu [37]. They stop testing when learning converges, but they also use other stopping criteria if learning does not converge.

V. Case Study

In the following, we will discuss some implementation-specific details concerning both learning and conformance checking of learned models.

A. Learning

The learning part was implemented in Scala using the Java-library LearnLib for active automata learning [31]. Most of the learning-related functionality was thus already implemented and we only had to implement application-specific components such as mappers, and a component responsible for the configuration of learning experiments.

1) Application-Specific Components: We had to implement the three components shown in the middle of Fig. 3, i.e. the adapter, the client interface, and the mapper. While adapter and client interface merely perform parsing, serialisation, and sending of packets, mappers specify a learning target.

As noted in Section IV, we used several sets of packet types with interesting dependencies within these sets as a basis for learning and consequently testing. For this purpose, we implemented seven different mappers, which all use the same client interface and thereby can be used interchangeably to test different aspects. Due to space limitations, we will describe only two of the seven implemented mappers.

Simple: The mapper *Simple* controls one client and offers seven inputs exercising only basic functionality such as the simplest forms of subscribing and publishing.

Two Clients with Retained Will: This mapper controls two clients, one of which sets a *will message* while connecting and which may close the TCP connection without properly disconnecting. The other client may subscribe to the topic to which the will message is published. More specifically, the will message is published as retained message which means that it is kept in the broker's state and sent whenever a client subscribes to the corresponding topic.

It is of course possible to define further mappers for learning other functionality. However, we do not aim at completely testing MQTT brokers. We rather aim at showing that our approach is an effective aid at finding errors and we assume that experiments performed with seven different mappers provide sufficient evidence for this purpose.

2) Configuration: In order to evaluate different learning algorithms, equivalence checking algorithms and MQTT implementations, we needed to implement a configuration component with which we can setup learning. However, a thorough comparison of algorithms is beyond the scope of the paper.

We used the TTT learning algorithm [30] for all experiments presented in the following as it performed best in our experiments. Furthermore, we used the random-walk equivalence oracle provided by LearnLib to perform equivalence queries. This equivalence oracle basically performs random tests to check whether the current hypothesis is correct. Although random testing is not well-suited to guarantee coverage of some kind it is a valid choice in our context. The main reason for this is the lack of scalability of more thorough methods like the W-method [16], [45] or the Wp-method [22]. These methods are computationally expensive in terms of runtime, even for only moderately complex models. In order to apply those, it would be necessary to use low depth values (difference in number of states between hypothesis and actual models) which limits their capability to find counterexamples. As a result, random testing was found to be better suited.

We used random walks with the following settings:

- probability of resetting the SUL: 0.05
- maximum number of steps: 10000
- reset step count after finding a counterexample: $true$

Additionally, there is a configuration parameter specifying an application-specific timeout on the receipt of messages.

B. Conformance Checking

In the following, we will describe our approach to checking conformance between two learned models. We actually check for equivalence and either output that the models are equivalent or present all found counterexamples to the user. This is accomplished through the application of bisimulation checks.

"On the Fly"-Check: We implemented this equivalence checking method in a way similar to the bisimulation check in [19]. For this purpose, we interpreted Mealy machines as LTSs whereby we interpreted input-output pairs labelling a transition in a Mealy machine as a single transition label in the LTS-interpretation.

In order to find counterexamples to equivalence we have to find *fail*-states in a product graph of the two considered models, created with respect to bisimulation. The graph contains states formed by pairs of states of both Mealy machines, additional fail-states, and transitions between those states. A transition between ordinary states is added for input-output pairs executable in both Mealy machines and a transition to a fail-state is added if a transition for some input-output pair is executable in only one of the Mealy machines, but not in the other. We thereby check for observation equivalence because we add a fail-state only if there is some input for which the two Mealy machines produce different outputs.

Since we consider deterministic Mealy machines, this check is simple to implement [19]. We implemented it via an implicit depth-first search for fail-states in the product graph. During this exploration, we collect all traces leading to fail-states and present them as counterexamples to the user. Since counterexamples are the only relevant information, we do not actually create the graph. It suffices to store visited states in order to avoid exploring some state twice.

Note that the straight-forward implementation may also miss some bugs. Consider a bug which merely produces wrong outputs but does not affect state transitions. In this case the bisimulation check will stop exploring at the wrong output and add the reached state to the visited states. If, however, it is necessary to explore the model beyond this state to find another bug, we may miss this bug. Hence, it is possible to miss *double faults* if both faults are reached by a single trace.

To circumvent this problem, we added the possibility to extend counterexamples further until either another difference is found or a visited state is reached. Actually, the exploration can be continued until a preset maximum number of differences along a trace has been found. With this extended exploration it is thus possible to find multiple counterexamples in cases where the standard exploration would have found only one. As a result, the effort required to analyse counterexamples is increased, but it may pay off if additional bugs are found.

The extended exploration did not uncover further bugs in our experiments, but led to a modification of the learning setup. Two models were incorrectly learned because of insufficient equivalence testing (equivalence query). This issue was detected by cross-checking with models of other implementations with extended exploration. Consequently, the number of steps for random equivalence testing has been increased.

TABLE I
TIMEOUT VALUES FOR RECEIVING MESSAGES.

Implementation	Timeout in Milliseconds
Apache ActiveMQ	300
emqttd	25
HBMQTT	100
Mosquitto	100
VerneMQ	300

C. Experiments

In the following, we will discuss our case study in more detail. We will start by discussing the basic setup. Afterwards we will describe some of the bugs and differences between models we found. In this context, we will consider difficulties and issues we faced as well as the manual effort required to classify counterexamples as failures.

Setup: We learned models of five freely available implementations of MQTT brokers, all of which are in active development at the time of writing this paper. The brokers are (included in):

- Apache ActiveMQ 5.13.3[2]
- emqttd 1.0.2 [3]
- HBMQTT 0.7.1 [4]
- Mosquitto 1.4.9 [5]
- VerneMQ 0.12.5p4 [6]

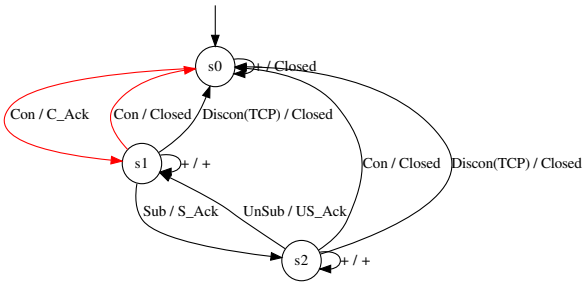
Since all brokers implement version 3.1.1 of MQTT, it was possible to perform all learning experiments in the same way with only minor adaptations. The adaptations basically amount to specifying application-specific timeouts for receiving packets. Table I shows the timeout values used for the different implementations. We found these values via experiments and note that they are neither optimal nor do large timeout values indicate poor performance in general. A broker requiring a large timeout may, e.g., provide excellent scalability to large numbers of connections which we did not test.

All experiments were performed with a Lenovo Thinkpad T450 with 16 GB RAM and an Intel Core i7-5600U CPU operating at 2.6 GHz and running Xubuntu Linux 14.04.

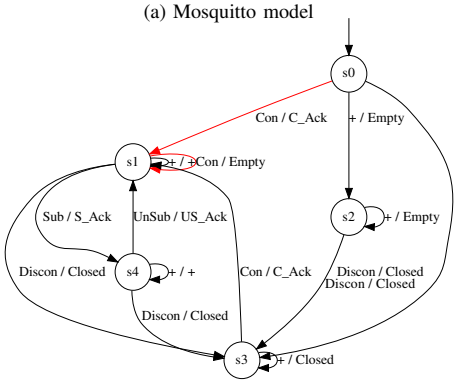
D. Bug Hunt

In the following, we will discuss our findings with respect to error detection in implementations. Altogether we found 17 bugs in all implementations combined whereby we did not find any in the Mosquitto broker. Additionally, we found two cases of unexpected non-determinism in two implementations which hindered learning and therefore are not included in the 17 bugs found. Finally, we found a part of the specification which strictly speaking none of the implementations implemented correctly. However, four of the implementations showed behaviour users would expect to see thus we consider these implementations to be correct. The last implementation

[2]http://activemq.apache.org/
[3]http://emqtt.io/
[4]https://github.com/beerfactory/hbmqtt
[5]http://mosquitto.org/
[6]https://vernemq.com/



(a) Mosquitto model



(b) HBMQTT model

Fig. 4. Models of two implementations learned with the *Simple* mapper, whereby some inputs and outputs have been combined (denoted by +).

on the other hand showed faulty behaviour. Hence, **we actually found 18 bugs**.

Four of the bugs correspond to issues already reported by other users. The remaining bugs were reported by us and are currently being reviewed or are already fixed by developers of the brokers. We will give some examples showing bugs we found and highlighting issues we faced.

1) Violations of the Specification: A simple example of a violation of the protocol specification can be found by considering the behaviour of the HBMQTT broker with respect to the functionality covered by the *Simple* mapper. Fig. 4 shows the models learned by observing the Mosquitto broker and the HBMQTT broker with abbreviated action labels. A counterexample to equivalence is $Connect \cdot Connect$, which is shown in red in both models. For Mosquitto we have the output $C\_Ack \cdot ConnectionClosed$ and for HBMQTT we have the output $C\_Ack \cdot Empty$, i.e. HBMQTT acknowledges the first connection attempt and ignores the second by not producing any output and it actually does not change its state as well.

The MQTT specification states that Mosquitto's behaviour is correct whereas HBMQTT behaves in an incorrect way [14]:

> A Client can only send the CONNECT Packet once over a Network Connection. The Server MUST process a second CONNECT Packet sent from a Client as a protocol violation and disconnect the Client [MQTT-3.1.0-2].

This is admittedly a rather simple example. However, we found also more subtle bugs. A strength with regard to error detection of our approach is that Mealy machines are input enabled. Therefore, we do not only learn and test the usual behaviour, but also exceptional cases. Using the mapper *Two Clients with Retained Will* we found an interesting sequence which uncovered bugs in both emqttd and ActiveMQ.

The sequence is as follows:

1) A client connects with client identifier `Client1`
2) A client connects with client identifier `Client2` with retained will message `bye` for topic `c2_will`
3) `Client2` disconnects unexpectedly (such that the will message is published)
4) `Client1` subscribes to `c2_will`
5) `Client1` subscribes to `c2_will`

The responses to the first two steps are delivered as expected, which are acknowledgements and the will message is sent to `Client1` in the fourth step which is also correct. In the fifth step, Mosquitto behaves differently from emqtt and ActiveMQ. While emqttd and ActiveMQ do not resend `bye` to `Client1`, the Mosquitto broker sends `bye` again.

The behaviour of ActiveMQ and emqtt is incorrect according to [MQTT-3.8.4-3] [14] which states that repeated subscription requests must replace existing subscriptions and that "any existing retained messages matching the Topic Filter MUST be re-sent".

2) Non-Determinism: An issue we faced during our experiments was non-deterministic behaviour with which Mealy-machine learning algorithms cannot cope. In the case of our setup which is based on LearnLib, an exception is thrown and learning is stopped as soon as non-deterministic behaviour is detected. Thus, we may waste test time in these cases. The only information we gain from such experiments is that non-determinism affects the experiments accompanied with two input/output sequences with the same inputs but different outputs, i.e. sequences witnessing non-determinism.

Non-determinism may result from several sources:

- learning setup
- time-dependent issues
- actual non-determinism displayed by implementations

In the first case, it is actually beneficial that learning stops, as the setup should not introduce non-determinism. It is likely to contain errors in this case. One issue related to time is the unknown time it takes for a broker to respond. In order to avoid non-deterministic behaviour in this regard, we implemented the aforementioned timeout on the receipt of messages.

We thus introduce imprecision to overcome time-related non-determinism. Considering that TCP is actually a reliable protocol and that the user datagram protocol (UDP) is often used in the IoT, time-related non-determinism is likely to be a more severe issue in other IoT protocols.

Implementations may also show truly non-deterministic behaviour, i.e. the repeated execution of some input sequence under the same conditions may lead to different results. Unfortunately, we cannot adapt our learning setup with reasonable effort to account for actual non-determinism. For this reason, we could not successfully perform 3 out of 35

learning experiments for the bug hunt. We actually evaluated further MQTT implementations in addition to those listed in Section V, which we excluded from the experiments because of non-deterministic behaviour. These additional implementations showed non-deterministic behaviour during learning with even the most simple mapper. Considering that, we conclude that learning-based verification would greatly benefit from being able to learn non-deterministic models.

3) Discussion: In the following, we will review our experiences using the proposed approach with special regard to manual effort. In this context, we will also recapitulate some already discussed issues affecting the required effort.

In the initial phase it requires some experimentation to define mappers with reasonable complexity, i.e. such that learning is possible in an acceptable amount of time. This usually does not require a substantial amount of human labour, but requires computation time as experiments have to be executed repeatedly with at least one implementation. This can probably best be compared to defining test-case specification scenarios, e.g., for testing with Spec Explorer [23].

However, we also spent a significant amount of time analysing suspicious traces, a task not needed in traditional model-based testing as requirements have to be formalised beforehand. In this context, we made the observation that bugs usually result in several counterexamples to equivalence. In addition to the standard equivalence check, we also used the extended bisimulation check to avoid missing bugs. This, however, required additional manual effort.

Consider for instance the first bug discussed and highlighted in Fig. 4. Essentially the same bug can be detected by analysing the counterexample $Connect \cdot Subscribe \cdot Connect$. Cross-checking the models in Fig. 4, we actually found 7 counterexamples with the bisimulation check and 24 with the bisimulation check performing extended exploration. All these counterexamples point to only two different bugs. An example in which the extended check does not cause any overhead is related to a bug of VerneMQ which causes the broker to not publish empty retained messages. Checking equivalence between a model of Mosquitto and a model of VerneMQ learned with a mapper not described in this paper finds 4 counterexamples with either of the checks.

At the current stage, we implemented a mechanism to manually define filters to hide counterexamples matching a specified pattern. Thus, it is possible to analyse a counterexample, find a bug and specify a pattern to exclude similar counterexamples. Coarse patterns may lead to bugs being undetected, therefore we did not use filters in our experiments.

However, a reduction of counterexamples or some kind of automated partitioning into equivalence classes of counterexamples may be crucial for a successful application of the approach to more complex systems. To implement such a technique it would, e.g., be possible to follow an approach similar to *MoreBugs* [28]. The MoreBugs method tries to infer a bug pattern from a failing test case and avoids testing the same pattern repeatedly. We could group counterexamples by matching them to inferred patterns and present only one counterexample per group to users. Especially parallel-composition-based pattern inference seems promising in our use case. Since Mealy machines are input-enabled, inputs result in self-loops in many states which causes counterexample traces to be interleaved with non-relevant inputs.

TABLE II
EXPERIMENTAL RESULTS OBTAINED BY LEARNING WITH THE MAPPER *Simple* WITH AN ALPHABET SIZE OF 7.

	ActiveMQ	emqttd	HBMQTT	Mosquitto	VerneMQ
# states	4	3	5	3	3
MQ time[s]	59.72	3.87	31.94	14.01	43.91
MQ # queries	88	59	110	56	57
CT time[s]	914.18	78.3	491.06	278.21	915.77
CT # queries	525	519	482	487	490
# equivalence queries	4	3	4	3	3

We noted in Section IV that it may be possible to learn an incorrect model if the equivalence oracle which is only an approximation provides a wrong answer. Therefore, we test counterexamples as stand-alone tests as well to see whether they are spurious. In this way, we actually create a regression test suite focused on previously detected bugs.

A more problematic scenario is that we may learn incompletely and thereby overlook erroneous behaviour. However, on the one hand we have seen that bugs usually result in several counterexamples which lowers the probability of missing bugs. On the other hand, testing is inherently incomplete, so there is always the possibility that we do not detect all bugs.

We conclude that it is possible to find non-trivial bugs in protocol implementations with reasonable effort despite necessary harsh abstraction. Testing more complex systems may, however, be hindered by the large number of counterexamples that need to be analysed. Tasks other than that have comparable counterparts in traditional model-based testing. It should be emphasised that the initial effort to setup a learning environment is relatively low due to the flexibility and ease of use of LearnLib [31].

E. Efficiency

We faced an issue during learning which is related to runtime. To illustrate the severity of this problem, Tables II and III show runtime measurement results for learning with the two described mappers.

The results include the number of states in the learned models, the time and number of queries needed for membership queries (*MQ time[s]* and *MQ # queries*), and the time and number of queries needed for conformance testing (*CT time[s]* and *CT # queries*). The number of queries for conformance testing reflects the actual number of tests carried out to check equivalence. In other words, this number denotes the number of conformance tests executed during the equivalence queries performed throughout learning. The number of equivalence queries represents the number of rounds of every learning experiment, i.e. the last row shows the number of hypotheses constructed by learning.

TABLE III
EXPERIMENTAL RESULTS OBTAINED BY LEARNING WITH THE MAPPER *Two Clients with Retained Will* WITH AN ALPHABET SIZE OF 9.

	ActiveMQ	emqttd	HBMQTT	Mosquitto	VerneMQ
# states	18	18	17	18	17
MQ time[s]	1855.55	167.32	557.14	641.89	1570.8
MQ # queries	732	735	640	730	625
CT time[s]	4787.92	481.36	2022.47	1612.59	4355.97
CT # queries	672	816	613	670	658
# equivalence queries	13	12	11	9	11

It can be observed that we actually deal with relatively simple models. The largest models have eighteen states and the larger of the two alphabets contains nine input symbols. Despite the possibility to learn much larger models with active automata learning, e.g., Merten et al. noted that they achieved to learn a system with over a million states [38], we still faced efficiency issues. This can be explained by considering the long runtime of individual tests/queries.

In our setting, tests may take several seconds since we wait up to 600 milliseconds for outputs from two clients in response to a single input (300 milliseconds per client). Thus, we see similar learning performance as when learning the TLS protocol [18]. However, unlike in the context of learning TLS, we do not stop testing once a connection is closed. Since there are persistent sessions and other related features of MQTT we also learn behaviour relevant to, e.g., session resumption.

The drastic influence of testing runtime can be seen in experiments performed with ActiveMQ and VerneMQ as they require the largest timeout value on the receipt of outputs. Even the simplest model of VerneMQ takes almost 16 minutes to learn (see Table II). The longest experiment, learning a model of ActiveMQ with the mapper *Two Clients Retained Will*, takes more than 110 minutes and resulted in a model with only eighteen states (see Table III). These high computation times for learning comparably simple models make apparent that there is a need to keep the number and length of queries to be executed as small as possible. This can, e.g., be achieved via domain-specific optimisations, heuristics and smart test selection [29], [42], or via algorithmic advantages [30].

VI. CONCLUSION

A. Summary

In this paper we presented a learning-based approach to semi-automatically detect failures of reactive systems. We evaluated the effectiveness of this approach by means of a case study. In total we found 18 faults in four out of five MQTT brokers.

More concretely, we learned abstract models of MQTT brokers. Based on that, we identified observable differences between the considered implementations in an automated manner. Since these differences are likely to show erroneous behaviour we inspected manually whether they show specification violations.

To the best of our knowledge, we presented the first such case study focusing on reactive systems implemented independently by open-source developers and it is the first attempt at model-based testing MQTT brokers. We showed that the proposed approach can be effective at detecting bugs without requiring any prior modelling. Additionally, we showed that interactions requiring a long time to complete can be an obstacle. It is a known fact that active automata learning shows efficiency problems while learning models with large input alphabets and state space [11]. This issue is especially problematic when dealing with systems with long and unknown response delays, a property exhibited by MQTT implementations.

While the approach can generally be applied to any type of reactive system for which there exist multiple implementations, it is especially well-suited to protocol testing because: (1) protocols can be modelled abstractly with low numbers of states, making active learning feasible, and (2) well-defined standards are likely to exist for common protocols. However, another possible application scenario is regression testing of reactive systems [29], i.e. a model of a new system version could be learned and checked for conformance to the model of a previous version.

B. Future Work

As noted before, we had to deal with non-determinism. It is actually common for complex reactive systems to behave non-deterministically so it may be worthwhile to investigate ways to learn non-deterministic models of reactive systems such as non-deterministic Mealy machines [33] or input-output transition systems (IOTSs) [48]. Both of the cited approaches unfortunately suffer from the fact that we can never be sure when we have seen all non-deterministic behaviours. In other words, we do not know how often we need to apply some sequence of inputs in order to see all possible responses. In practice, assumptions have to be made with which we can derive bounds on the number of required repetitions of some input sequence. Through the requirement of repeated executions, however, the computation time increases which is already an issue.

Alternatively, output non-determinism [33] may be resolved by learning probabilistic rather than non-deterministic models. There exist promising passive learning approaches, e.g., based on state merging, which infer (probabilistic) models from samples obtained prior to learning [15], [17]. Such methods have already been investigated in a verification context [35].

We also observed that there exists time-dependent behaviour and that such behaviour is likely to play a more crucial role in other IoT protocols using means of communication less reliable than TCP. As a result, there is a need for an investigation of appropriate methods to infer models of timed systems in this area. There exist approaches for this task, in an active setting for learning event-recording automata [24], [25], [34] and in a passive setting for learning timed automata with a single clock [46], [47]. Both approaches place restrictions on the expressiveness of models they consider. The former, however, is less restricted at the expense of higher worst-case complexity. Hence, it would be interesting to examine practical

limits of these approaches and whether they could be improved in terms of performance or learnability. Verwer et al. [46] identified two intriguing starting points for future research concerning learnability of more expressive timed automata: (1) they showed that in general timed automata with multiple clocks cannot be efficiently identified. However, specific classes of timed automata might be efficiently identifiable. (2) Timed automata with n clocks can actually be represented by the intersection of n timed automata with only one clock. Hence, a possible approach would be to learn multiple one-clock timed automata in a first step and combine them in a second step.

A possible extension to the MQTT case study would be to infer models of client implementations as well and to verify properties of the composition of clients and brokers. Such an approach has been followed by Fiterău-Broştean et al. for analysing TCP implementations [21].

ACKNOWLEDGMENT

This work was supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". The authors would like to thank the LEAD project members Masoud Ebrahimi, Franz Pernkopf, Franz Röck, and Tobias Schrank for fruitful discussions, and Florian Lorber, Richard Schumi and the anonymous reviewers for their feedback. Additionally, we would like to thank the developers of LearnLib.

REFERENCES

[1] F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F. W. Vaandrager, "Learning register automata with fresh value generation," in *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, ser. Lecture Notes in Computer Science, M. Leucker, C. Rueda, and F. D. Valencia, Eds., vol. 9399. Springer, 2015, pp. 165–183.

[2] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. W. Vaandrager, "Automata learning through counterexample guided abstraction refinement," in *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, ser. Lecture Notes in Computer Science, D. Giannakopoulou and D. Méry, Eds., vol. 7436. Springer, 2012, pp. 10–27.

[3] F. Aarts, B. Jonsson, and J. Uijen, "Generating models of infinite-state communication protocols using regular inference with abstraction," in *Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings*, ser. Lecture Notes in Computer Science, A. Petrenko, A. da Silva Simão, and J. C. Maldonado, Eds., vol. 6435. Springer, 2010, pp. 188–204.

[4] F. Aarts, H. Kuppens, J. Tretmans, F. W. Vaandrager, and S. Verwer, "Learning and testing the bounded retransmission protocol," in *Proceedings of the Eleventh International Conference on Grammatical Inference, ICGI 2012, University of Maryland, College Park, USA, September 5-8, 2012*, ser. JMLR Proceedings, J. Heinz, C. de la Higuera, and T. Oates, Eds., vol. 21. JMLR.org, 2012, pp. 4–18.

[5] B. K. Aichernig and C. C. Delgado, "From faults via test purposes to test cases: On the fault-based testing of concurrent systems," in *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, ser. Lecture Notes in Computer Science, L. Baresi and R. Heckel, Eds., vol. 3922. Springer, 2006, pp. 324–338.

[6] B. K. Aichernig, E. Jöbstl, and S. Tiran, "Model-based mutation testing via symbolic refinement checking," *Science of Computer Programming*, vol. 97, pp. 383–404, 2015.

[7] B. K. Aichernig and M. Tappler, "Symbolic input-output conformance checking for model-based mutation testing," *Electr. Notes Theor. Comput. Sci.*, vol. 320, pp. 3–19, 2016.

[8] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[9] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

[10] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen, "On the correspondence between conformance testing and regular inference," in *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, ser. Lecture Notes in Computer Science, M. Cerioli, Ed., vol. 3442. Springer, 2005, pp. 175–189.

[11] T. Berg, B. Jonsson, M. Leucker, and M. Saksena, "Insights to Angluin's learning," *Electr. Notes Theor. Comput. Sci.*, vol. 118, pp. 3–18, 2005.

[12] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 535–552.

[13] H. Brandl, M. Weiglhofer, and B. K. Aichernig, "Automated conformance verification of hybrid systems," in *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010*, J. Wang, W. K. Chan, and F. Kuo, Eds. IEEE, 2010, pp. 3–12.

[14] E. by Andrew Banks and R. Gupta, "MQTT Version 3.1.1," OASIS Standard, October 2014, latest version: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[15] R. C. Carrasco and J. Oncina, "Learning stochastic regular grammars by means of a state merging method," in *Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, ser. ICGI '94. London, UK, UK: Springer-Verlag, 1994, pp. 139–152.

[16] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. Softw. Eng.*, vol. 4, no. 3, pp. 178–187, May 1978.

[17] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. New York, NY, USA: Cambridge University Press, 2010.

[18] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, J. Jung and T. Holz, Eds. USENIX Association, 2015, pp. 193–206.

[19] J. Fernandez and L. Mounier, ""On the Fly" verification of behavioural equivalences and preorders," in *Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July, 1-4, 1991, Proceedings*, ser. Lecture Notes in Computer Science, K. G. Larsen and A. Skou, Eds., vol. 575. Springer, 1991, pp. 181–191.

[20] P. Fiterău-Broştean, R. Janssen, and F. W. Vaandrager, "Learning fragments of the TCP network protocol," in *Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings*, ser. Lecture Notes in Computer Science, F. Lang and F. Flammini, Eds., vol. 8718. Springer, 2014, pp. 78–93.

[21] ——, "Combining model learning and model checking to analyze TCP implementations," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds., vol. 9780. Springer, 2016, pp. 454–471.

[22] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models," *IEEE Trans. Softw. Eng.*, vol. 17, no. 6, pp. 591–603, Jun. 1991.

[23] W. Grieskamp and N. Kicillof, "A schema language for coordinating construction and composition of partial behavior descriptions," in *Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, ser. SCESM '06. New York, NY, USA: ACM, 2006, pp. 59–66.

[24] O. Grinchtein, B. Jonsson, and M. Leucker, "Learning of event-recording automata," *Theor. Comput. Sci.*, vol. 411, no. 47, pp. 4029–4054, 2010.

[25] O. Grinchtein, B. Jonsson, and P. Pettersson, "Inference of event-recording automata using timed decision trees," in *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, ser. Lecture Notes

in Computer Science, C. Baier and H. Hermanns, Eds., vol. 4137. Springer, 2006, pp. 435–449.

[26] A. Groce, D. A. Peled, and M. Yannakakis, "Adaptive model checking," in *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, ser. Lecture Notes in Computer Science, J. Katoen and P. Stevens, Eds., vol. 2280. Springer, 2002, pp. 357–370.

[27] F. Howar, B. Steffen, and M. Merten, "Automata learning with automated alphabet abstraction refinement," in *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, 2011, pp. 263–277.

[28] J. Hughes, U. Norell, N. Smallbone, and T. Arts, "Find more bugs with QuickCheck!" in *Proceedings of the 11th International Workshop on Automation of Software Test, AST@ICSE 2016, Austin, Texas, USA, May 14-15, 2016*, C. J. Budnik, G. Fraser, and F. Lonetti, Eds. ACM, 2016, pp. 71–77.

[29] H. Hungar, O. Niese, and B. Steffen, "Domain-specific optimization in automata learning," in *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, ser. Lecture Notes in Computer Science, W. A. H. Jr. and F. Somenzi, Eds., vol. 2725. Springer, 2003, pp. 315–327.

[30] M. Isberner, F. Howar, and B. Steffen, "The TTT algorithm: A redundancy-free approach to active automata learning," in *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, ser. Lecture Notes in Computer Science, B. Bonakdarpour and S. A. Smolka, Eds., vol. 8734. Springer, 2014, pp. 307–322.

[31] ——, "The open-source LearnLib - A framework for active automata learning," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Kroening and C. S. Pasareanu, Eds., vol. 9206. Springer, 2015, pp. 487–495.

[32] "Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1, ISO/IEC 20922:2016," International Organization for Standardization, Geneva, CH, Standard, Jun. 2016.

[33] A. Khalili and A. Tacchella, "Learning nondeterministic Mealy machines," in *Proceedings of the 12th International Conference on Grammatical Inference, ICGI 2014, Kyoto, Japan, September 17-19, 2014.*, ser. JMLR Proceedings, A. Clark, M. Kanazawa, and R. Yoshinaka, Eds., vol. 34. JMLR.org, 2014, pp. 109–123.

[34] S. Lin, É. André, J. S. Dong, J. Sun, and Y. Liu, "An efficient algorithm for learning event-recording automata," in *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, ser. Lecture Notes in Computer Science, T. Bultan and P. Hsiung, Eds., vol. 6996. Springer, 2011, pp. 463–472.

[35] H. Mao, Y. Chen, M. Jaeger, T. Nielsen, K. Larsen, and B. Nielsen, "Learning deterministic probabilistic automata from a model checking perspective," *Machine Learning*, pp. 1–45, 2016.

[36] T. Margaria, O. Niese, H. Raffelt, and B. Steffen, "Efficient test-based model generation for legacy reactive systems," in *Ninth IEEE International High-Level Design Validation and Test Workshop 2004, Sonoma Valley, CA, USA, November 10-12, 2004*. IEEE Computer Society, 2004, pp. 95–100.

[37] K. Meinke and M. A. Sindhu, "Incremental learning-based testing for reactive systems," in *Tests and Proofs - 5th International Conference, TAP 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*, ser. Lecture Notes in Computer Science, M. Gogolla and B. Wolff, Eds., vol. 6706. Springer, 2011, pp. 134–151.

[38] M. Merten, F. Howar, B. Steffen, and T. Margaria, "Automata learning with on-the-fly direct hypothesis construction," in *Leveraging Applications of Formal Methods, Verification, and Validation - International Workshops, SARS 2011 and MLSC 2011, Held Under the Auspices of ISoLA 2011 in Vienna, Austria, October 17-18, 2011. Revised Selected Papers*, ser. Communications in Computer and Information Science, R. Hähnle, J. Knoop, T. Margaria, D. Schreiner, and B. Steffen, Eds., vol. 336. Springer, 2011, pp. 248–260.

[39] O. Niese, "An integrated approach to testing complex systems," Ph.D. dissertation, Dortmund University of Technology, 2003.

[40] D. A. Peled, M. Y. Vardi, and M. Yannakakis, "Black box checking," *Journal of Automata, Languages and Combinatorics*, vol. 7, no. 2, pp. 225–246, 2002.

[41] M. Shahbaz and R. Groz, "Inferring Mealy machines," in *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, ser. Lecture Notes in Computer Science, A. Cavalcanti and D. Dams, Eds., vol. 5850. Springer, 2009, pp. 207–222.

[42] W. Smeenk, J. Moerman, F. W. Vaandrager, and D. N. Jansen, "Applying automata learning to embedded control software," in *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, ser. Lecture Notes in Computer Science, M. Butler, S. Conchon, and F. Zaïdi, Eds., vol. 9407. Springer, 2015, pp. 67–83.

[43] J. Tretmans, "Test generation with inputs, outputs and repetitive quiescence," *Software - Concepts and Tools*, vol. 17, no. 3, pp. 103–120, 1996.

[44] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, Aug. 2012.

[45] M. P. Vasilevskii, "Failure diagnosis of automata," *Cybernetics*, vol. 9, no. 4, pp. 653–665, 1973.

[46] S. Verwer, M. de Weerdt, and C. Witteveen, "The efficiency of identifying timed automata and the power of clocks," *Inf. Comput.*, vol. 209, no. 3, pp. 606–625, 2011.

[47] ——, "Efficiently identifying deterministic real-time automata from labeled data," *Machine Learning*, vol. 86, no. 3, pp. 295–333, 2012.

[48] M. Volpato and J. Tretmans, "Approximate active learning of nondeterministic input output transition systems," *ECEASST*, vol. 72, 2015.

[49] M. Weiglhofer and F. Wotawa, ""On the fly" input output conformance verification," in *Proceedings of the IASTED International Conference on Software Engineering*, ser. SE '08. Anaheim, CA, USA: ACTA Press, 2008, pp. 286–291.