Learning Abstracted Non-deterministic Finite State Machines

Andrea Pferscher, Bernhard K. Aichernig

Learning Abstracted Non-Deterministic Finite State Machines

Andrea Pferscher(✉) and Bernhard K. Aichernig[0000−0002−3484−5584]

Institute of Software Technology, Graz University of Technology, Austria
{apfersch,aichernig}@ist.tugraz.at

Abstract. Active automata learning gains increasing interest since it gives an insight into the behavior of a black-box system. A crucial drawback of the frequently used learning algorithms based on Angluin's L^* is that they become impractical if systems with a large input/output alphabet are learned. Previous work suggested to circumvent this problem by abstracting the input alphabet and the observed outputs. However, abstraction could introduce non-deterministic behavior. Already existing active automata learning algorithms for observable non-deterministic systems learn larger models if outputs are only observable after certain input/output sequences. In this paper, we introduce an abstraction scheme that merges akin states. Hence, we learn a more generic behavioral model of a black-box system. Furthermore, we evaluate our algorithm in a practical case study. In this case study, we learn the behavior of five different Message Queuing Telemetry Transport (MQTT) brokers interacting with multiple clients.

Keywords: Active automata learning · Model inference · Non-deterministic finite state machines · MQTT.

1 Introduction

The origin of automata learning dates back to the seventies and eighties, when Gold [9] and Angluin [4] introduced algorithms to learn behavioral models of black-box systems. Later, in the seminal work of Peled et al. [17], automata learning proofed itself as a valuable tool to test black-box systems. Today automata learning reveals security vulnerabilities in TLS [20] or DTLS [8].

The applicability of automata learning, however, suffers from two main problems: (1) automata learning becomes infeasible for systems with a large input and output alphabet and (2) many systems behave non-deterministically, e.g. due to timed behavior or stochastic decisions. One promising solution for the first problem was proposed by Aarts et al. [2]. They presented an abstraction technique that introduces a more generic view on the system by creating an abstract and, therefore, smaller input and output alphabet. However, a too coarse view on the system creates non-determinism, which leads back to the second problem. In the literature, several work on learning-based testing [23,8,3] stress the problem of observing non-deterministic behavior during learning. However, we already find

learning algorithms [7,16,11] for observable non-deterministic reactive systems. To make them applicable, we have to provide an input and output alphabet that establishes observable non-determinism, i.e. systems where an input may lead to different states, but we can observe which state has been chosen.

In this paper, we propose an active learning algorithm for *observable non-deterministic finite state machines (ONFSMs)* that addresses both problems of active automata learning. The first problem is solved via a mapper that provides a more generic view on the system under test (SUT) via an abstract input/output alphabet. The non-deterministic behavior is handled by our learning algorithm for ONFSMs. We show that an abstracted input/output alphabet does not necessarily decrease the state space of a non-deterministic systems sufficiently. To overcome the state-space problem, we introduce a new abstraction technique that defines equivalence classes for outputs and merges akin states of the model.

We evaluated our proposed learning algorithm on the MQTT protocol, which is a publish/subscribe protocol that is frequently used in the Internet of Things (IoT). This protocol defines the communication between clients and a broker, where the broker manages the messages of the clients. Tappler et al. [23] showed that automata learning is an effective method to test the behavior of different MQTT brokers. However, in their work they only considered an interaction with one or two clients to keep the learning feasible. Furthermore, they stress that the non-deterministic behavior hampered the learning procedure. In this paper, we show that our algorithm for observable non-deterministic systems makes learning-based testing of the MQTT protocol in a multi-client setup feasible.

Our contribution comprises four aspects. (1) We analyze the challenges of learning an abstracted system that behaves non-deterministically. (2) We propose a new abstraction technique that manages these challenges. (3) We introduce a new active learning algorithm that integrates our proposed abstraction technique and, therefore, learns a more abstract model of a non-deterministic system. (4) We show the applicability of our algorithm in a case study and compare it to existing algorithms for deterministic finite state machines (FSMs).

Structure. Section 2 explains the modeling formalism and active automata learning. In Sect. 3, we introduce our learning algorithm including our novel abstraction technique. Our case study on five different MQTT brokers is presented in Sect. 4. Section 5 discusses related work. Finally, Sect. 6 concludes the paper.

2 Preliminaries

2.1 Finite State Machines

An *observable non-deterministic finite state machine (ONFSM)* is a 5-tuple $\mathcal{M} = \langle \Sigma_I, \Sigma_O, Q, q_0, h \rangle$ where Σ_I is the finite set of input symbols, Σ_O is the finite set of output symbols, Q is the finite set of states, q_0 is the initial state, and $h \subseteq Q \times \Sigma_I \times \Sigma_O \times Q$ is the transition relation.

We denote a transition in the ONFSM by $q \xrightarrow{\mathsf{i/o}} q'$, where $(q, i, o, q') \in h$. The assumed ONFSMs are *input enabled*, i.e., there exists for every pair (q, i), where

$q \in Q$ and input $i \in \Sigma_I$, at least one output $o \in \Sigma_O$ and successor state $q' \in Q$. Further, we call the finite state machine *observable* if there exists for any triple (q, i, o) at most one q' such that (q, i, o, q') belongs to h. Hence, in fact, we have a state transition function $\delta : Q \times I \times O \to Q$, that maps a triplet (q, i, o) to a unique state q'. Note that learning algorithms, like the ones used in the tool **LearnLib** [10], commonly require deterministic systems. We define an FSM as deterministic if there exists for each input at most one output and succeeding state. In the literature, deterministic FSM are also known as Mealy machines.

Let $s \in (\Sigma_I \times \Sigma_O)^*$ be an input/outputs sequence, with the corresponding input projection $s_I \in \Sigma_I^*$ and output projection $s_O \in \Sigma_O^*$. We write ϵ for the empty sequence and $s \cdot s'$ for the concatenation of two sequences, where $s, s' \in (\Sigma_I \times \Sigma_O)^*$. We also define a single input/output pair $(i, o) \in (\Sigma_I \times \Sigma_O)$ as a sequence and, therefore, $s \cdot (i, o)$ is also defined. The function $pre(s)$ returns the prefixes of s, including the sequence s. Let $s^+ \in (\Sigma_I \times \Sigma_O)^+, s_I^+ \in \Sigma_I^+, s_O^+ \in \Sigma_O^+$ denote non-empty sequences. For convenience, we write $s \cdot (s_I^+, s_O^+)$, defining a sequence s that is concatenated with a sequence of input/output pairs.

2.2 Active Automata Learning

In active automata learning, we gain understanding about a system by actively questioning the SUT. Many active automata learning algorithms build on the seminal work of Angluin [4]. Her proposed algorithm (L^*) learns a deterministic finite automaton (DFA) that accepts a regular language. The learning procedure is based on the minimally adequate teacher (MAT) framework, which comprises a *teacher* and a *learner* component. The learner asks the teacher either a *membership query* or an *equivalence query*. The former one answers if a word is part of the language, whereas the second one checks if a hypothesis, i.e. a proposed DFA, represents the regular language. The equivalence query is either answered with *yes* or with a counterexample that shows an evidence to the non-conformance of the proposed hypothesis and the SUT. All the answers from the queries are saved in an *observation table*, which is then used to construct a hypothesis.

Angluin's L^* has been extended for different types of systems, including algorithms for Mealy machines [12,15,21]. In these algorithms for learning Mealy machines, membership queries are replaced by *output queries*. There, instead of asking whether a word is in the language, an input string is provided. The teacher executes this input string on the SUT and responds the observed output string. The observed outputs are then saved in the observation table.

The previously mentioned learning algorithms can only handle deterministic SUTs. Tackling the problem that systems behave non-deterministic due to various reasons, e.g. ignoring timed behavior, L^*-based learning algorithms [7,16,11] for ONFSMs were proposed. The algorithms for ONFSMs follow the idea of the Mealy machine learning algorithms, but instead of considering just one possible output for an input, all possible outputs are saved in the observation table.

One major drawback of the Angluin style algorithms is that they do not scale for a large input/output alphabet, since the number of required queries significantly grows with the increasing size of the used alphabet. To overcome this

problem, Aarts et al. [2] proposed an abstraction technique that reduces the size of the used input/output alphabet of the learning algorithm. This abstraction technique introduces a mapper component that translates the communication between the learner and the SUT in both directions.

A mapper is an 8-tuple $\mathcal{A} = \langle \Sigma_I, \Sigma_O, R, r_0, \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, \Delta, \nabla \rangle$, where Σ_I and Σ_O are the disjoint sets of concrete inputs and outputs, R is the set of mapper states, $r_0 \in R$ is the initial state, $\Sigma_I^{\mathcal{A}}$ and $\Sigma_O^{\mathcal{A}}$ are the finite disjoint sets of abstract inputs and outputs, $\Delta : R \times (\Sigma_I \cup \Sigma_O) \to R$ is the state transition function, and $\nabla : (R \times \Sigma_I \to \Sigma_I^{\mathcal{A}}) \cup (R \times \Sigma_O \to \Sigma_O^{\mathcal{A}})$ is the abstraction function. The mapper takes abstract inputs and translates them to concrete inputs, which can then be executed on the SUT. In return, the mapper observes concrete outputs from the SUT and returns the corresponding abstracted outputs to the learner. On each received input/output pair, the mapper updates its internal state.

A large input/output alphabet is not the only challenge that limits the feasibility of active automata learning. Berg et al. [5] state that the real bottleneck of active automata learning is the equivalence oracle since the presence of such an oracle is highly improbable. To make active learning practically applicable, tools like LearnLib [10] substitute the equivalence oracle by conformance testing.

3 Method

In this section, we describe our active learning algorithm for abstracted ONFSMs. Firstly, we introduce our basic learning setup. Secondly, we explain the two different levels of abstraction and, thirdly, we discuss the application of our abstraction mechanism on the learning algorithm.

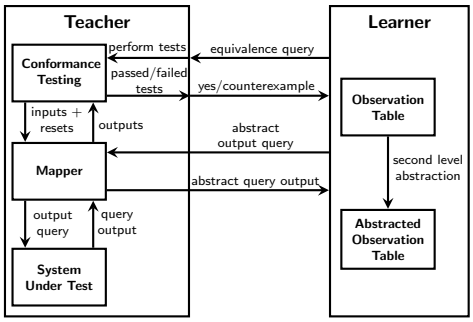

Fig. 1. Angluin's [4] traditional learning framework is extended by a mapper component and an abstracted observation table.

3.1 Learning Setup

Our proposed learning algorithm for ONFSMs is based on an Angluin-style active learning setup. For this, we define a *learner* that queries a *teacher*. Figure 1 shows the components of our learning algorithm. The teacher comprises three components: (1) conformance testing, (2) a mapper and (3) the SUT. (1) The aim of conformance testing is to make the equivalence check feasible. For this, we assume that the learned model conforms to the SUT if a finite set of test cases passes. (2) The mapper is based on the idea of Aarts et al. [2], where the mapper translates a (possibly infinite) alphabet to a finite alphabet that is used by the learning algorithm to create an abstracted model. The difference to Aarts' proposed mapping concept is that we do not require an abstracted alphabet that assures that the learned model behaves deterministically. This gives us the possibility to create an even more abstract view on the SUT. (3) The SUT represents an interface, where concrete inputs can be executed and the corresponding concrete outputs can be observed by the mapper. This allows a black-box view on the SUT.

The difference to other active learning setups is that the learner uses an additional abstraction mechanism for the observation table. To avoid confusion between the abstraction done by the mapper and the abstraction of the observation table, we distinguish two levels of abstraction. We denote the abstraction by the mapper as first level abstraction and the abstraction of the observation table is denoted as second level abstraction.

3.2 First Level Abstraction

The first level abstraction comprises the mapper component of the teacher that translates abstract inputs to concrete inputs and concrete outputs to abstract outputs. The aim of mapping is to decrease the size of the alphabet to make the learning algorithm feasible. However, Aarts et al. [2] stress that a too abstract alphabet creates non-determinism, which is not suitable for many learning algorithms. We circumvent the problem of creating a deterministic alphabet by using a learning algorithm for non-deterministic systems. Still, the abstracted alphabet must not violate the assumption of observable non-deterministic outputs. If the abstraction is too coarse, we have to refine the abstracted outputs by introducing new outputs that make the model of the SUT observable non-deterministic.

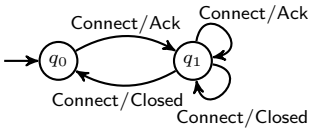

Fig. 2. Abstracted non-deterministic Mealy machine of the MQTT connection protocol

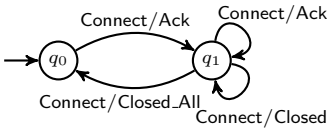

Fig. 3. The ONFSM of the MQTT connection protocol

Example 1 (Multi-Client Connection Mapper). Our goal is to learn a model of an MQTT broker that interacts with several clients. For simplicity, here we only consider the connection procedure of the protocol. The clients can send a connection request to the broker. A client gets disconnected if the client is already connected. The concrete inputs are $\Sigma_I = \{\mathsf{Connect}(id) | id \in \mathbb{N}\}$ and the concrete outputs $\Sigma_O = \{\mathsf{Ack}(id), \mathsf{Closed}(id) | id \in \mathbb{N}\}$, where $id \in \mathbb{N}$ uniquely identifies a client by an integer. Therefore, the set of concrete inputs and outputs is infinite. To build a finite model of this connection protocol for multiple-clients, we have to define a mapper that abstracts the input/output alphabet. The mapper for this example has the abstracted inputs $\Sigma_I^{\mathcal{A}} = \{\mathsf{Connect}\}$ and abstracted outputs $\Sigma_O^{\mathcal{A}} = \{\mathsf{Ack}, \mathsf{Closed}\}$. The abstracted non-deterministic model is depicted in Fig. 2. We distinguish the states where no client (q_0) or more than one client (q_1) is connected. In the state where more than one client is connected, further clients can connect or disconnect. Since the Mealy machine in Fig. 2 is not observable non-deterministic, we have to refine the abstracted outputs to make the states distinguishable. For this, we add another abstract output $\mathsf{Closed\_All}$, capturing the observation when the last client gets disconnected, to $\Sigma_O^{\mathcal{A}}$ which makes the automaton observable non-deterministic. The refined model is depicted in Fig. 3.

3.3 Second Level Abstraction

The first level abstraction enables an abstracted view and supports the feasibility of the learning algorithm. However, due to the iterative state exploration of L^*,

Table 1. Observation table of the multi-client connection protocol of Example 1.

$\Gamma \backslash E$	Connect
ϵ	Ack
(Connect, Ack)	Ack, Closed_All
(Connect, Ack)(Connect, Ack)	Ack, Closed
(Connect, Ack)(Connect, Closed_All)	Ack

Table 2. Abstracted observation table of Table 1.

$\Gamma \backslash E$	Connect
ϵ	Ack
(Connect, Ack)	Ack, Closed
(Connect, Ack)(Connect, Ack)	Ack, Closed
(Connect, Ack)(Connect, Closed_All)	Ack

the first level abstraction might not be enough to keep the number of states low. To overcome this problem, we introduce a second level abstraction. The aim of this abstraction level is to decrease the state space by defining equivalent outputs. This is done by the definition of equivalence classes for outputs, which then replace the observations saved in the observation table. The mapping of the equivalence classes is a surjective function $\mu : \Sigma_O^{\mathcal{A}} \to \Sigma_O^{\mathcal{A}'}$ that maps an abstracted output of $\Sigma_O^{\mathcal{A}}$ to an equivalence class of $\Sigma_O^{\mathcal{A}'}$.

Example 2 (Multi-Client Connection Equivalence Classes). Even though we abstracted the alphabet in Example 1, learning algorithms for ONFSMs, e.g. L_N [7], would learn a model similar to the one shown in Fig. 4. The problem is that these algorithms count the number of clients, and create a new state for each client. To avoid a large model, we want to merge akin states. For this, we define for the outputs $\Sigma_O^{\mathcal{A}} = \{\mathsf{Ack}, \mathsf{Closed}, \mathsf{Closed\_All}\}$ the following equivalence classes $\Sigma_O^{\mathcal{A}'} = \{\mathsf{Ack}, \mathsf{Closed}\}$. The function μ for the second level abstraction of this example is defined as follows $\mu : \{\mathsf{Ack}\} \to \mathsf{Ack}, \{\mathsf{Closed}, \mathsf{Closed\_All}\} \to \mathsf{Closed}$.

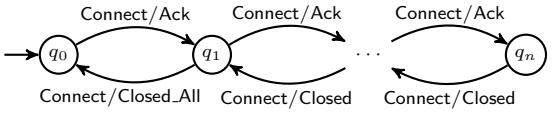

Fig. 4. Learned model of Example 1 without second level abstraction, which has $n+1$ states for n clients.

In our learning algorithm, the learner maintains an extended observation table. The classic observation table $\mathcal{T}$ is defined as a triplet $\langle \Gamma, E, T \rangle$, where $\Gamma \subset (\Sigma_I^{\mathcal{A}} \times \Sigma_O^{\mathcal{A}})^*$ contains sequences of inputs and outputs, $E \subset \Sigma_I^{\mathcal{A}^+}$ is a suffixed closed set of input sequences, and T is a mapping $\Gamma \times E \to 2^{\Sigma_O^{\mathcal{A}^+}}$ containing first level abstract outputs of the power set of $\Sigma_O^{\mathcal{A}^+}$. The extension of the classical observation table $\mathcal{T}^{\mathcal{A}'}$ is a 5-tuple $\langle \Gamma, E, T, T^{\mathcal{A}'}, \mu \rangle$ where $T^{\mathcal{A}'}$ is a mapping $\Gamma \times E \to 2^{\Sigma_O^{\mathcal{A}'^+}}$ to the second level abstracted outputs and $\mu : \Sigma_O^{\mathcal{A}} \to \Sigma_O^{\mathcal{A}'}$ is the second level abstraction function.

Example 3 (Multi-Client Connection Observation Table). Table 1 shows an intermediate observation table in the learning procedure of the system depicted in Fig. 3. Table 1 is then abstracted with the equivalence classes defined in Example 2. The abstracted observation table is presented in Table 2.

3.4 Learning Algorithm

In the following, we introduce our learning algorithm for ONFSMs. Especially, we explain the integration of the second level abstraction in the L^*-based learning setup and how this abstraction technique generates a more generic hypothesis.

Algorithm 1 Learning algorithm using an abstracted observation table

Input: input alphabet $\Sigma_I^{\mathcal{A}}$, equivalence class mapping μ, teacher Π
Output: ONFSM $\mathcal{H}$
1: $\langle \Gamma, E, T\rangle \leftarrow$ InitTable($\Sigma_I^{\mathcal{A}}$)
2: cex $\leftarrow \epsilon$
3: **do**
4: $\quad\langle \Gamma, E, T\rangle \leftarrow$ FillTable($\langle \Gamma, E, T\rangle, \Pi$)
5: $\quad\mathcal{T}^{\mathcal{A}'} \leftarrow$ AbstractTable($\langle \Gamma, E, T\rangle, \mu$)
6: $\quad$**while** $\neg$IsClosedConsistentComplete($\mathcal{T}^{\mathcal{A}'}$) **do**
7: $\quad\quad\Gamma \leftarrow$ MakeClosed($\mathcal{T}^{\mathcal{A}'}$)
8: $\quad\quad\langle \Gamma, E, T\rangle \leftarrow$ FillTable($\langle \Gamma, E, T\rangle, \Pi$)
9: $\quad\quad\mathcal{T}^{\mathcal{A}'} \leftarrow$ AbstractTable($\langle \Gamma, E, T\rangle, \mu$)
10: $\quad\quad\Gamma \leftarrow$ CompleteQueries($\mathcal{T}^{\mathcal{A}'}$)
11: $\quad\quad\langle \Gamma, E, T\rangle \leftarrow$ FillTable($\langle \Gamma, E, T\rangle, \Pi$)
12: $\quad\quad\mathcal{T}^{\mathcal{A}'} \leftarrow$ AbstractTable($\langle \Gamma, E, T\rangle, \mu$)
13: $\quad\quad E \leftarrow$ MakeConsistent($\mathcal{T}^{\mathcal{A}'}$)
14: $\quad\quad\langle \Gamma, E, T\rangle \leftarrow$ FillTable($\langle \Gamma, E, T\rangle, \Pi$)
15: $\quad\quad\mathcal{T}^{\mathcal{A}'} \leftarrow$ AbstractTable($\langle \Gamma, E, T\rangle, \mu$)
16: $\quad\mathcal{H} \leftarrow$ ConstructHypothesis($\mathcal{T}^{\mathcal{A}'}$)
17: $\quad$cex $\leftarrow$ ConformanceTest($\mathcal{H}$)
18: $\quad$**if** cex $\neq \epsilon$ **then**
19: $\quad\quad\langle \Gamma, E, T\rangle \leftarrow$ AddCex(cex, Π)
20: **while** cex $\neq \epsilon$
21: **return** $\mathcal{H}$

Algorithm 1 describes the procedure performed by the learner to learn an ONFSM. For this, the learner requires the abstract input alphabet $\Sigma_I^{\mathcal{A}}$ of the SUT, the second level abstraction mapping μ, and the teacher Π which provides access to the mapper and SUT. In Line 1 and 2 we start with the initialization of the observation table and the counterexample (cex). Remember that the classic observation table is a triplet $\mathcal{T} = \langle \Gamma, E, T\rangle$. Γ is a prefixed-closed set of sequences, which can be written as $\Gamma = \Gamma_S \cup \Gamma_P$ where $\Gamma_S \cap \Gamma_P = \emptyset$. Γ_S contains the sequences that identify the states of the ONFSM and $\Gamma_P = \Gamma_{P_S} \cup \Gamma_{P'}$, where $\Gamma_{P_S} \subseteq \Gamma_S \cdot (\Sigma_I^{\mathcal{A}} \times \Sigma_O^{\mathcal{A}})$, $\Gamma_{P'} \subset \Gamma \cdot (\Sigma_I^{\mathcal{A}} \times \Sigma_O^{\mathcal{A}})$ and $\Gamma_{P_S} \cap \Gamma_{P'} = \emptyset$. We initialize $\mathcal{T}$ by adding the empty sequence ϵ to Γ_S and the input alphabet $\Sigma_I^{\mathcal{A}}$ to E.

After the initialization we fill the observation table $\mathcal{T}$ (Line 4) by performing output queries to the teacher Π. The mapper translates the first level abstracted inputs of $\Sigma_I^{\mathcal{A}}$ to concrete inputs of Σ_I and the observed concrete outputs of Σ_O to first level abstracted outputs $\Sigma_O^{\mathcal{A}}$. Every time we fill the table with the query outputs we extend the observation table $\mathcal{T} = \langle \Gamma, E, T\rangle$ to the abstract observation table $\mathcal{T}^{\mathcal{A}'} = \langle \Gamma, E, T, T^{\mathcal{A}'}, \mu\rangle$ by translating observed outputs with μ as explained in Sect. 3.3. Note that the second level abstraction only changes the mapping $T^{\mathcal{A}'}$; Γ, E and T stay unchanged.

After the abstraction in Line 5 we check in Line 6 if the abstracted observation table $\mathcal{T}^{\mathcal{A}'}$ is *closed*, *consistent* and *complete*. To the best of our knowledge, this part is different to other learning algorithms.

Closedness First, we check if the abstracted table $\mathcal{T}^{\mathcal{A}'}$ is *closed*. This check is similar to the closed check proposed by El-Fakih et al. [7], however, instead of checking the values in the mapping T, we check if the second level abstracted outputs in $T^{\mathcal{A}'}$ are equal. For this, we denote that two rows $\gamma, \gamma' \in \Gamma$ in the abstracted table are equal, i.e. $\gamma \cong_{\mathcal{A}'} \gamma'$, iff $\forall e \in E, T^{\mathcal{A}'}(\gamma, e) = T^{\mathcal{A}'}(\gamma', e)$. The abstract observation table $\mathcal{T}^{\mathcal{A}'}$ is not closed if there exists a $\gamma' \in \Gamma_P$ such that no $\gamma \in \Gamma_S$ fulfills $\gamma \cong_{\mathcal{A}'} \gamma'$.

Consistency Unlike the other learning algorithms for ONFSMs, we have to check if our abstract observation table $\mathcal{T}^{\mathcal{A}'}$ is *consistent*. The consistency check is necessary due to queries that are added through the abstraction mechanism and due to the counterexample processing which we discuss later in this section. Our check is based on the consistency check for Mealy machines introduced by

Table 3. Observation table of Example 1 after the first output queries.

$\Gamma \backslash E$	Connect
ϵ	Ack
(Connect, Ack)	Ack, Closed_All

Table 4. Abstraction of Table 3 using the mapping defined in Example 2

$\Gamma \backslash E$	Connect
ϵ	Ack
(Connect, Ack)	Ack, Closed

Niese [15]. We define that $\mathcal{T}^{\mathcal{A}'}$ is consistent if for every $\gamma, \gamma' \in \Gamma$, where $\gamma \cong_{\mathcal{A}'} \gamma'$ holds, no input/output pair $(i, o) \in (\Sigma_I^{\mathcal{A}} \times \Sigma_O^{\mathcal{A}})$ exists where $\gamma \cdot (i, o) \ncong_{\mathcal{A}'} \gamma' \cdot (i, o)$.

Completeness The *completeness* check is different to other active learning algorithms. Here we check, if we have added all necessary sequences to construct a hypothesis. Since the final hypothesis contains the first level abstracted outputs, but is constructed based on the second level abstraction outputs, it may be necessary to add additional sequences to Γ. These sequences are required to identify the target state of all observed input/output pairs. For this, we define that two rows $\gamma, \gamma' \in \Gamma$ in the classic observation table are equal, denoted by $\gamma \cong \gamma'$, iff $\forall e \in E, T(\gamma, e) = T(\gamma', e)$. For each $\gamma \in \Gamma_S$ we select each $\gamma' \in \Gamma_P$ where $\gamma \cong_{\mathcal{A}'} \gamma'$ but $\gamma \ncong \gamma'$ holds. We then check if Γ contains all sequences $pre(\gamma' \cdot (e, s_O^+))$ where $e \in E$ and $T(\gamma, e) \neq T(\gamma', e)$, and $s_O^+ \in T(\gamma', e)$. If no such sequence is required or all required sequences already exist, $\mathcal{T}^{\mathcal{A}'}$ is complete.

If either the abstracted table $\mathcal{T}^{\mathcal{A}'}$ is not *closed* or not *consistent* or not *complete*, we continue our procedure in Line 7. Here, we make $\mathcal{T}^{\mathcal{A}'}$ closed if necessary. To make $\mathcal{T}^{\mathcal{A}'}$ closed, we move every row $\gamma' \in \Gamma_P$ to Γ_S, where $\forall \gamma \in \Gamma_S : \gamma \ncong_{\mathcal{A}'} \gamma'$ holds. Informally, we move every row from Γ_P to Γ_S where the set of abstracted outputs for all $e \in E$ is unique. This introduces a new state in our hypothesis. If we have found such a $\gamma' \in \Gamma_P$, we have to add all observed outputs in the mapping T from the row γ' to Γ_{P_S}. For this, we concatenate γ' with each observed input/output pair in that row, i.e. for each $i \in \Sigma_I^{\mathcal{A}}$ we add for each observed output sequence $o \in T(\gamma', i)$ the concatenation $\gamma' \cdot (i, o)$ to Γ_{P_S}. Afterwards, we fill the table $\mathcal{T}$ according to the updated Γ in Line 8 and in the next line we construct again $\mathcal{T}^{\mathcal{A}'}$.

Example 4 (Closed Abstracted Observation Table). Table 3 shows the observation table after the first output queries. We abstract the table using the mapping function μ introduced in Example 2. The result of the abstraction is shown in Table 4. According to Algorithm 1 we then check if the abstracted table is closed. Since the row (Connect, Ack) is in Γ_P and the produced outputs are not in Γ_S, Table 4 is not closed. To make the table closed, we move (Connect, Ack) to Γ_S and add the corresponding sequences of the observed outputs to Γ_P. The result including the additionally performed output queries is shown in Table 1. We then again abstract this table – shown in Table 2 – and check if the abstracted table is closed. Our closed check on Table 2 is now satisfied. Note that we do not bother that Table 1 is not closed.

After making the abstract table closed and performing the required queries to fill the table, we add in Line 10 all necessary queries to make the table complete. For this, we select for all $\gamma \in \Gamma_S$ every $\gamma' \in \Gamma_P$ where $\gamma \cong_{\mathcal{A}'} \gamma'$. If then $\gamma \ncong \gamma'$, we select the input sequence $e \in E$ where $T(\gamma, e) \neq T(\gamma', e)$ and the output sequence where the observed outputs are different, i.e. $s_O^+ \in T(\gamma', e) \setminus T(\gamma, e)$.

Table 5. Final observation table of the connection protocol of Example 1. The input Connect is abbreviated by Conn.

$\Gamma \backslash E$	Conn
ϵ	Ack
(Conn, Ack)	Ack, Closed_All
(Conn, Ack)(Conn, Ack)	Ack, Closed
(Conn, Ack)(Conn, Closed_All)	Ack
(Conn, Ack)(Conn, Ack)(Conn, Closed)	Ack, Closed_All

Table 6. Final abstracted observation table generated from Table 5 The input Connect is abbreviated by Conn.

$\Gamma \backslash E$	Conn
ϵ	Ack
(Conn, Ack)	Ack, Closed
(Conn, Ack)(Conn, Ack)	Ack, Closed
(Conn, Ack)(Conn, Closed_All)	Ack
(Conn, Ack)(Conn, Ack)(Conn, Closed)	Ack, Closed

The sequences $pre(\gamma' \cdot (e, s_O^+))$ are added to $\Gamma_{P'}$, if the sequence is not already part of Γ. Therefore, $\Gamma_{P'}$ only contains sequences that are added to fulfill the completeness of the abstracted observation table.

Example 5 (Complete Abstracted Observation Table). We continue the learning procedure of Example 4 by checking if the observation table is complete. For this, we consider Table 1 and the abstracted Table 2. We see that the row (Connect, Ack) is equal to (Connect, Ack), (Connect, Ack) in the abstracted table. However, in Table 1 we see that the outputs Closed_All and Closed are different. Hence, we add the sequence (Connect, Ack), (Connect, Ack), (Connect, Closed) to the classic observation table (Table 5). After making Table 5 complete, we abstract the table (Table 6). Table 6 is now closed and complete.

If the table is *closed* and *complete*, we make in Line 13 the abstracted observation table *consistent*. For each two rows $\gamma, \gamma' \in \Gamma$ where $\gamma \cong_{\mathcal{A}'} \gamma'$ and an input/output pair $(i, o) \in (\Sigma_I^{\mathcal{A}} \times \Sigma_O^{\mathcal{A}})$ where $\gamma \cdot (i, o) \ncong_{\mathcal{A}'} \gamma' \cdot (i, o)$ exists, we add $i \cdot e$ to E, where $e \in E$ and $T(\gamma \cdot (i, o), e) \neq T(\gamma' \cdot (i, o), e)$.

When the table is closed, complete and consistent, we construct a hypothesis from our abstract observation table in Line 16 of Algorithm 1. Since we consider two different levels of abstraction in our observation table, we have to modify the classical construction approach from El-Fakih et al. [7]. Algorithm 2 describes the process of generating an ONFSM $\mathcal{M}$ from the abstract observation table $\mathcal{T}^{\mathcal{A}'}$. The initialization process in Line 2 and 3 of Algorithm 2 is equal to other L^*-style algorithms. For this, the empty sequence ϵ indicates the initial state q_0 of $\mathcal{M}$. In addition, the states are defined by the rows of Γ_S. However, due to the second level abstraction, not all observed outputs are represented in the row $\gamma_S \in \Gamma_S$. To consider these outputs in the hypothesis, we have to check

Algorithm 2 Hypothesis construction

Input: abstract obs. table $\mathcal{T}^{\mathcal{A}'} = \langle \Gamma = \Gamma_S \cup \Gamma_P, E, T, T^{\mathcal{A}'}, \mu \rangle$, input alphabet $\Sigma_I^{\mathcal{A}}$
Output: ONFSM $\mathcal{M} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, Q, q_0, h \rangle$
1: **function** ConstructHypothesis
2: $q_0 \leftarrow \epsilon \in \Gamma_S$
3: $Q \leftarrow \Gamma_S$
4: **for all** $\gamma_S \in \Gamma_S$ **do**
5: $Q_{\gamma_S} \leftarrow \{\gamma | \gamma \in \Gamma \wedge \gamma \cong_{\mathcal{A}'} \gamma_S\}$
6: **for all** $\gamma \in Q_{\gamma_S}$ **do**
7: **for all** $i \in \Sigma_I^{\mathcal{A}}$ **do**
8: **for all** $o \in T(\gamma, i)$ **do**
9: $\gamma' \leftarrow \gamma \cdot (i, o)$
10: **if** $\gamma' \in \Gamma$ **then**
11: $h \leftarrow h \cup (\gamma_S, i, o, \gamma'_S)$ where $\gamma'_S \in \Gamma_S \wedge \gamma'_S \cong_{\mathcal{A}'} \gamma'$
return $\mathcal{M}$

also the observations in the mapping T. For this, we filter in Line 5 the equal rows of γ_S in Γ according to the second level abstraction and then add all transitions with the corresponding observations in the Lines 6 to 11. In Line 9 we concatenate the currently considered sequence γ with the observed input/output pair. If this sequence is part of Γ, we have to find the correct destination state of the transition. For this, we select the equal state according to the second level abstraction in Γ_S. In Line 11, we add the new transition to the transitions h.

Example 6 (Hypothesis Construction). Table 5 and Table 6 are used to construct a hypothesis of Example 1. The closed, complete and consistent Table 6 identifies the states and Table 5 contains all observed outputs. Γ_S includes two states $\{q_0, q_1\}$. From the initial state q_0 we create one transition $q_0 \xrightarrow{\mathsf{Connect/Ack}} q_1$. In the second state we consider three different outputs, $\{\mathsf{Ack}, \mathsf{Closed}, \mathsf{Closed\_All}\}$, where Closed and Closed_All belong to the equivalence class Closed. Therefore, we create three different edges $q_1 \xrightarrow{\mathsf{Connect/Closed\_All}} q_0$, $q_1 \xrightarrow{\mathsf{Connect/Closed}} q_1$ and $q_1 \xrightarrow{\mathsf{Connect/Ack}} q_1$. The hypothesis is equal to the ONFSM shown in Fig. 3.

Conformance Testing After the construction of the hypothesis (Algorithm 1, Line 16), we have to check in Line 17 whether our constructed hypothesis conforms to the SUT. Usually, in automata learning we assume that two systems are equal if the behavior of the learned hypothesis and the SUT are equal. For an ONFSM we could say that two systems are equal if all observable traces are equal. However, due to our second level abstraction, the learned model has a more generic behavior than the SUT. Therefore, the trace equivalence assumption does not hold. Instead we have to check trace inclusion, i.e., every trace produced by the SUT must also be observable in our learned hypothesis. Let $\mathcal{I}$ be our SUT and $\mathcal{H}$ the learned hypothesis. Further we define the function $\textsc{Traces}(\mathcal{M})$ that returns all observable traces, starting at the initial state q_0 of an ONFSM $\mathcal{M}$. The generated traces contain input and output symbols on the first level abstraction. Formally, we can define the conformance relation as follows:

$$\textsc{Traces}(\mathcal{I}) \subseteq \textsc{Traces}(\mathcal{H}). \tag{1}$$

This relations implies that the more generic the hypothesis the better the conformance. However, a completely generic model would not be useful, e.g. for further testing purposes. The learning of a too generic hypothesis is prevented by the mere consideration of observed outputs from the observation table. Therefore, the hypothesis only includes outputs that can be observed on the SUT.

According to the framework depicted in Fig. 1, the teacher either responds *yes* if the conformance relation in Eq. 1 is fulfilled or reports a counterexample. The counterexample is a trace that contains an output that is not included in our hypothesis $\mathcal{H}$. Note that since the model is input-enabled only outputs may lead to the violation of trace inclusion. We replace the conformance oracle by conformance testing to make the conformance check feasible. For this, a finite number of test cases is generated and then executed on the SUT. Since the test cases contain inputs and outputs on the first level abstraction, we again need a mapper that abstracts and concretizes inputs and outputs for the execution on the SUT. A test case *passes* if the trace generated by the SUT is also observable

in the hypothesis, otherwise the test case *fails*. Therefore, the teacher answers with *yes*, if all traces *pass*, otherwise a failing trace is returned. In Algorithm 1 Line 17 the return of an empty sequence ϵ is equal to the answer *yes*. A practical implementation of the conformance testing approach is discussed in Sect. 4.

If no counterexample is found our algorithm terminates and returns the current hypothesis in Line 21, otherwise we add the counterexample to the observation table in Line 19. Note that every returned counterexample is cut off after the first non-conforming input/output pair.

The adding of a counterexample in Line 19 distinguishes two different types of counterexamples. The first type reveals an unobserved output in a state of the hypothesis where an output of the same equivalence class is also observable. To check if the counterexample belongs to this type, we execute all input/output pairs except the last one of the sequence on $\mathcal{H}$. This leads us to the *failing* state prior to the wrong observation. We then check if executing the remaining input generates an output of the same equivalence class. If the output is in the same equivalence class, we want to add the missing input/output transition to the state. In this case, we add any not already existing prefix of the found counterexample to Γ_P. The second type of a counterexample reveals an input/output pair, where no equivalent output is observable in the failing state. In this case, we perform the counterexample processing as proposed by El-Fakih et al. [7], since we want to introduce a new state in our hypothesis. This is done by firstly removing the longest matching prefix $\gamma \in \Gamma$ of the counterexample and secondly adding the suffixes of the remaining input sequence of the found counterexample to E. In both cases, we then jump back to Line 3 and continue to fill the observation table by asking membership queries. This procedure is repeated until no counterexample can be found.

4 Case Study

We evaluated the Message Queuing Telemetry Transport (MQTT) protocol. MQTT defines a publish/subscribe protocol, where clients can subscribe to topics and publish messages to a topic. Tappler et al. [23] already presented work on model-based testing of the MQTT protocol via active automata learning. However, in their work they only consider one or two clients interacting with the broker. Due to the proposed abstraction technique used in our learning algorithm for ONFSMs, we can deal with a multi-client setup. In this section, we first discuss technical aspects of the implementation of our learning algorithm and then present the results of the case study on five different MQTT brokers.

We weaken the *all-weather assumption* [13] for the observation of outputs, and assume like other learning algorithms for non-deterministic systems [16,11,26] that all outputs are observable after performing a query n times. Regarding the challenge that not all outputs are observable at once, we discuss three technical aspects of our implementation: (1) repeated execution of output queries, (2) stopping criterion for output queries, and (3) shrinking of the observation table.

(1) We need to repeat an output query since the abstraction done by the mapper introduces non-determinism. The mapper takes an abstract input and

randomly chooses an input from a set of corresponding concrete inputs. Therefore, the observed outputs may differ. The number of repetitions $n_q \in \mathbb{N}$ depends on the SUT, e.g. for the multi-client setup on MQTT we executed each query at least as often as the number of used clients.

(2) Repeating every output query each time the table is filled leads to a vast amount of queries. In practice, performing queries on the SUT can be expensive, e.g. due to the latency of the response. Thus, we want to reduce the number of performed queries. In our implementation, we extended each cell of the observation table by a score $s \in \mathbb{R}_{\geq 0}$ that indicates how often the outputs in the cell change. The value of s is between 0.0 and 1.0, and each time we perform the output queries and no new output is observed, s increases by a value $s_i \in \mathbb{R}_{\geq 0}$. If at least one new output is observed, the score decreases by $s_d \in \mathbb{R}_{\geq 0}$. Note that s cannot increase above 1.0 and decrease below 0.0. If $s = 1.0$, we do not perform the output query again. The initial value of s and the values of s_i and s_d depend on the SUT and how often output queries are repeated in general.

(3) The size of the observation table significantly influences the runtime of the learning algorithm. Our abstraction techniques keep the size of the observation table small. However, due to the assumed non-determinism, it may happen that rows become equal after repeating output queries. For this, we remove equal rows in Γ_S and the corresponding suffixes in Γ_P. The shrinking of the table is performed before the closedness, consistency and completeness check.

In Sect. 3.4, we explained that the equivalence oracle is replaced by conformance testing. Due to our conformance relation defined in Eq. 1, we cannot check if the outputs of all traces are equivalent, since our hypothesis is more generic than the SUT. We generate input sequences by randomly walking through the hypothesis model. The random walk selects outgoing transitions from the current state uniformly at random and append the input of the transition to the input sequence. This input sequence is then executed once on the SUT to generate an input/output sequence. We then check if the generated sequence is observable on our hypothesis. When we execute the input sequence on the SUT, we use the mapper to translate the abstracted inputs to concrete inputs. Since the number of possible concrete inputs is extremely large and executing all possible inputs would be infeasible, we limit the number of executed test cases to a finite number $n_{\text{test}} \in \mathbb{N}$. Due to the assumption of non-deterministic behavior, we may observe different sequences if we repeat the execution on the SUT. Instead of repeating each input sequence several times, we assume that the generation via random walk reflects the non-deterministic behavior sufficiently.

In this case study, we learned the ONFSMs of MQTT brokers. In the MQTT protocol, a client can connect to a broker. A connected client can subscribe to a topic and/or publish a message on a topic, which is then forwarded by the broker to the subscribed clients. The task of the MQTT broker is to handle the connection/disconnection of the clients and to forward publish-messages to subscribed clients. The five learned MQTT brokers are listed in Table 7. All of them implement the current MQTT standard version 5.0 [14]. To communicate with the different brokers, we used our own client implementation.

Table 7. Learning setup and results of our case study on MQTT brokers

Broker	**ejabberd**[1]	**EMQ X**[2]	**HiveMq**[3]	**Mosquitto**[4]	**VerneMQ**[5]
Version	20.3	v4.0.0	2020.2	1.6.8	1.10.0
Timeout	100	50	100	50	50
# Output Queries	18 315	18 375	15 950	13 975	14 800
# Equivalence Checks	1	1	1	1	1
Runtime (h)	11.28	5.48	9.04	3.98	4.30

We used an akin learning setup for all five MQTT brokers. The goal was to learn the behavior of the MQTT broker that interacts with multiple clients. In our experiments, we used five clients. We defined the abstract input alphabet by $\Sigma_I^A = \{\mathsf{Connect}, \mathsf{Disconnect}, \mathsf{Subscribe}, \mathsf{Unsubscribe}, \mathsf{Publish}\}$. The mapper translates each abstract input to a concrete input for one of the five clients. Furthermore, the clients can subscribe to one topic and publish a message to this topic. For the second level abstraction we defined the following mapping μ for the equivalence classes $\{\mathsf{Closed}, \mathsf{Closed\_all}, \mathsf{Closed\_Unsuback\_all}\} \to \mathsf{Closed}$, $\{\mathsf{Unsuback}, \mathsf{Unsuback\_all}\} \to \mathsf{Unsuback}$, and all other outputs map to an equally named singleton equivalence class. To make the approach feasible for a large number of clients, our mapper saves translations from the abstract sequences in Γ to sequences with concrete input/output values, i.e., we cache concrete input/output sequences of the sequences in Γ. For each of these saved translations, we repeat the output query times the number of clients. Using this querying technique, we assume that we do not frequently observe new outputs when we repeat the output queries. Therefore, we set the change indicator $s = 0.9$, $s_i = 0.2$ and $s_d = 0.1$. Furthermore, we assume that 2 000 input/output sequences sufficiently represent this system. Thus, we set $n_{\text{test}} = 2\,000$.

Figure 5 represents the learned ONFSM of an MQTT broker interacting with multiple clients. The symbol "+" represents a wildcard for several inputs or outputs that are not critical for the behavior system and, therefore, are skipped to keep the model clear. The three states of the MQTT model distinguish between the state where no client is connected (q_0), at least one client is connected, but no client is subscribed (q_1) and at least one client is connected and subscribed (q_2). Note that Fig. 5 represents a valid model for all setups where more than two clients interact with the MQTT broker.

Table 7 shows the learning setup and the results of our case study. The first three rows of the table define the MQTT broker, the tested version and the required timeout on responses. The timeout defines the duration how long (in milliseconds) the socket listens for incoming messages. An equal behavior of the five brokers could only be achieved by the adaption of the timeouts. Selecting a too low timeout, we may not receive all messages in time. Our proposed learning algorithm can deal with such a non-deterministic behavior. For example, when we learn the *HiveMQ* broker with a timeout of 50, we receive messages from old sessions or some messages are not delivered in time. Therefore, we set the

[1] https://www.ejabberd.im/
[2] https://www.emqx.io/
[3] https://www.hivemq.com/
[4] https://mosquitto.org/
[5] https://vernemq.com/

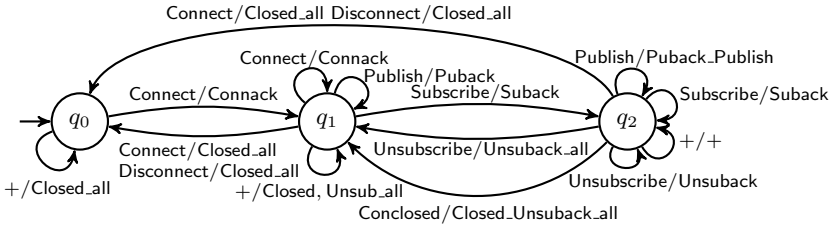

Fig. 5. The ONFSM of an MQTT broker that interacts with multiple-clients.

timeout for the brokers individually. The last three rows of Table 7 represent the learning results which state the number of performed output queries and equivalence queries, and the required time to learn the MQTT broker. The timeout directly affects the runtime of the learning algorithm, which can be seen in the experiments for *ejabberd* and *HiveMQ*. The experiments were conducted with a MacBook Pro 2019 with an Intel Quad-Core i5 operating at 2.4 GHz and with 8 GB RAM. All learned models correspond to the ONFSM presented in Fig. 5.

Furthermore, we compared our algorithm with a deterministic learning algorithm. For this, we learn a deterministic model of the MQTT broker that interacts with five clients. The experiment was performed with the tool **LearnLib** [10] using the improved version of L^* proposed by Rivest and Schapire [19]. We only learned the model for the Mosquitto broker with a timeout of 50, all other brokers of Table 7 showed non-deterministic behavior during learning. The learning of the Mosquitto broker for five clients took more than two days (58.29 h). The final model had 243 states and 6 075 transitions. The model size shows that even if we learn a small part of a rather simple system the complexity of the system increases significantly in a multi-client setup. To learn the deterministic model, 151 900 output queries and one equivalence query were required. These results stress that our algorithm enables a faster learning-based testing technique and provides a simpler model of a large system.

5 Related Work

Our proposed learning algorithm for ONFSM is based on Angluin's L^* [4]. In the literature, the majority of learning algorithms for non-deterministic systems [6,7,16,11,25,26] also follows the same principle. Bollig et al. [6] introduced a learning algorithm for residual finite-state automata, which are a subclass of non-deterministic finite automata. Using this modeling formalism, they can represent a deterministic finite-state automaton by a more succinct non-deterministic automaton. However, they assume that the SUT behaves deterministically.

All Angluin-based algorithms for ONFSMs [7,16,11] basically follow the approach that was first proposed by El-Fakih et al. [7]. Their algorithm is based on the *all-weather assumption*, i.e. all possible outputs are observable at once. Pacharoen et al. [16] revise this assumption by assuming that all outputs are observed, after executing a query $n \in \mathbb{N}$ times. Kahlili and Tacchella [11] also considered more practical aspects and learned the non-deterministic model of TFTP server. However, as explained in Sect. 3 these algorithms might not scale well when an alphabet is abstracted by a mapper component.

Volpato and Tretmans [25,26] introduced two learning algorithms for non-deterministic input/output labelled transition systems. In their first work [25], they posed the all-weather assumption on the observation of outputs and assume an Angluin-style equivalence oracle. In their second work [26], they weakened these assumptions and proposed a conformance relation based on the **ioco**-theory [24] for an over- and under-approximation of the SUT. In contrast to our algorithm, they try to improve the observation table by increasing it, which leads to more states like in the previously mentioned algorithms for ONFSMs.

Petrenko and Avellaneda [18] proposed an algorithm for ONFSM that is not based on Angluin's MAT framework. Instead of a teacher they create a hypothesis using a SAT solver. However, the algorithm is based on the assumption that the final number of states of the ONFSM must be known in advance.

Tappler et al. [22] presented an L^*-based learning algorithm for Markov decision processes. For this, they also considered the observed frequencies of stochastic events in their models. Their algorithm rather learns the stochastic behavior instead of the non-deterministic behavior like our algorithm.

6 Conclusion

We presented an active automata learning algorithm for observable non-deterministic finite state machines. Unlike previous work on L^*-based algorithms for ONFSMs, our algorithm learns a more abstract model of the SUT by merging akin states. For this, we defined two levels of abstraction and explained how to combine them. Firstly, we showed how the mapper component proposed by Aarts et al. [2] can be used in an active learning setup for ONFSMs. Secondly, to further improve the scalability of our learning algorithm, we introduced a new abstraction scheme for the observation table based on equivalence classes for outputs. This abstraction technique made it possible to learn a more abstract model of a system that has a large input/output alphabet. We showed the applicability of our implementation by learning the ONFSMs of different MQTT brokers that interact with multiple clients.

Our proposed learning algorithm offers a more feasible technique to apply learning-based testing in a broader field of applications. For future work, it would be interesting to show a more elaborated case study, e.g. considering more features of the MQTT protocol, so that our abstract model can reveal unexpected behavior of the SUT. Currently, the abstraction refinement relies on expert knowledge, but we assume that an automatic abstraction is possible. Aarts et al. [1] proposed an automatic abstraction refinement for the abstracted input/output alphabet that ensures deterministic learning. Following a similar idea, we could refine the outputs for the first level-abstraction to ensure observable non-determinism. Regarding the second level abstraction, an equivalence measurement for states could support the definition of equivalence classes.

Acknowledgment. This work is supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". We would like to thank student Jorrit Stramer for the implementation of the MQTT client.

References

1. Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F.W.: Automata learning through counterexample guided abstraction refinement. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7436, pp. 10–27. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_4
2. Aarts, F., Jonsson, B., Uijen, J., Vaandrager, F.W.: Generating models of infinite-state communication protocols using regular inference with abstraction. Formal Methods Syst. Des. **46**(1), 1–41 (2015). https://doi.org/10.1007/s10703-014-0216-x
3. Aichernig, B.K., Burghard, C., Korosec, R.: Learning-based testing of an industrial measurement device. In: Badger, J.M., Rozier, K.Y. (eds.) NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11460, pp. 1–18. Springer (2019). https://doi.org/10.1007/978-3-030-20652-9_1
4. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
5. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the correspondence between conformance testing and regular inference. In: Cerioli, M. (ed.) Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3442, pp. 175–189. Springer (2005). https://doi.org/10.1007/978-3-540-31984-9_14
6. Bollig, B., Habermehl, P., Kern, C., Leucker, M.: Angluin-style learning of NFA. In: Boutilier, C. (ed.) IJCAI 2009, Pasadena, CA, USA, July 11-17, 2009. pp. 1004–1009 (2009), `http://ijcai.org/Proceedings/09/Papers/170.pdf`
7. El-Fakih, K., Groz, R., Irfan, M.N., Shahbaz, M.: Learning finite state models of observable nondeterministic systems in a testing context. In: 22nd IFIP International Conference on Testing Software and Systems, Natal, Brazil. pp. 97–102 (2010), `https://hal.inria.fr/hal-00953395`
8. Fiterau-Brostean, P., Jonsson, B., Merget, R., de Ruiter, J., Sagonas, K., Somorovsky, J.: Analysis of DTLS implementations using protocol state fuzzing. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2523–2540. USENIX Association (2020), `https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean`
9. Gold, E.M.: System identification via state characterization. Automatica **8**(5), 621–636 (1972). https://doi.org/10.1016/0005-1098(72)90033-7
10. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib - A framework for active automata learning. In: Kroening, D., Pasareanu, C.S. (eds.) CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 487–495. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_32
11. Khalili, A., Tacchella, A.: Learning nondeterministic mealy machines. In: Clark, A., Kanazawa, M., Yoshinaka, R. (eds.) Proceedings of the 12th International Conference on Grammatical Inference, ICGI 2014, Kyoto, Japan, September 17-19, 2014. JMLR Workshop and Conference Proceedings, vol. 34, pp. 109–123. JMLR.org (2014), `http://proceedings.mlr.press/v34/khalili14a.html`

12. Margaria, T., Niese, O., Raffelt, H., Steffen, B.: 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. pp. 95–100. IEEE Computer Society (2004). https://doi.org/10.1109/HLDVT.2004.1431246
13. Milner, R.: A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92. Springer (1980). https://doi.org/10.1007/3-540-10235-3
14. OASIS message queuing telemetry transport (MQTT) TC. Standard, Organization for the Advancement of Structured Information Standards, Burlington, MA, USA (2019), `https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html`
15. Niese, O.: An integrated approach to testing complex systems. Ph.D. thesis, Technical University of Dortmund, Germany (2003), `https://d-nb.info/969717474/34`
16. Pacharoen, W., Aoki, T., Bhattarakosol, P., Surarerks, A.: Active learning of nondeterministic finite state machines. Mathematical Problems in Engineering **2013**, 1–11 (2013). https://doi.org/10.1155/2013/373265
17. Peled, D.A., Vardi, M.Y., Yannakakis, M.: Black box checking. J. Autom. Lang. Comb. **7**(2), 225–246 (2002). https://doi.org/10.25596/jalc-2002-225
18. Petrenko, A., Avellaneda, F.: Learning and adaptive testing of nondeterministic state machines. In: 19th IEEE International Conference on Software Quality, Reliability and Security, QRS 2019, Sofia, Bulgaria, July 22-26, 2019. pp. 362–373. IEEE (2019). https://doi.org/10.1109/QRS.2019.00053
19. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Inf. Comput. **103**(2), 299–347 (1993). https://doi.org/10.1006/inco.1993.1021
20. de Ruiter, J., Poll, E.: Protocol state fuzzing of TLS implementations. In: Jung, J., Holz, T. (eds.) 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015. pp. 193–206. USENIX Association (2015), `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter`
21. Shahbaz, M., Groz, R.: Inferring mealy machines. In: Cavalcanti, A., Dams, D. (eds.) FM 2009, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5850, pp. 207–222. Springer (2009). https://doi.org/10.1007/978-3-642-05089-3_14
22. Tappler, M., Aichernig, B.K., Bacci, G., Eichlseder, M., Larsen, K.G.: L^*-based learning of Markov decision processes. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11800, pp. 651–669. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_38
23. Tappler, M., Aichernig, B.K., Bloem, R.: Model-based testing IoT communication via active automata learning. In: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017. pp. 276–287. IEEE Computer Society (2017). https://doi.org/10.1109/ICST.2017.32
24. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Softw. Concepts Tools **17**(3), 103–120 (1996)
25. Volpato, M., Tretmans, J.: Active learning of nondeterministic systems from an ioco perspective. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. Lecture Notes in Computer Science, vol. 8802, pp. 220–235. Springer (2014). https://doi.org/10.1007/978-3-662-45234-9_16
26. Volpato, M., Tretmans, J.: Approximate active learning of nondeterministic input output transition systems. ECEASST **72** (2015). https://doi.org/10.14279/tuj.eceasst.72.1008