Journal of Automated Reasoning (2019) 63:1103–1134
https://doi.org/10.1007/s10817-018-9486-0

Efficient Active Automata Learning via Mutation Testing

Bernhard K. Aichernig[1] · Martin Tappler[1]

Received: 8 October 2018 / Accepted: 10 October 2018 / Published online: 25 October 2018

Abstract
System verification is often hindered by the absence of formal models. Peled et al. proposed black-box checking as a solution to this problem. This technique applies active automata learning to infer models of systems with unknown internal structure. This kind of learning relies on conformance testing to determine whether a learned model actually represents the considered system. Since conformance testing may require the execution of a large number of tests, it is considered the main bottleneck in automata learning. In this paper, we describe a randomised conformance testing approach which we extend with fault-based test selection. To show its effectiveness we apply the approach in learning experiments and compare its performance to a well-established testing technique, the partial W-method. This evaluation demonstrates that our approach significantly reduces the cost of learning. In multiple experiments, we reduce the cost by at least one order of magnitude.

Keywords Conformance testing · Mutation testing · FSM-based testing · Active automata learning · Minimally adequate teacher framework

1 Introduction

Since Peled et al. [26] have shown that active automata learning can provide models of black-box systems to enable formal verification, this kind of learning has turned into an active area of research in formal methods. Due to its close relation to testing, model learning has also gained popularity in the testing community, leading to the development of various combinations with model-based testing [3]. Active learning of automata in the minimally adequate teacher (MAT) framework, as introduced by Angluin [5], assumes the existence of a teacher. This teacher must be able to answer two types of queries, *membership* and *equivalence queries*. The former corresponds to a single test of the system under learning (SUL) to check whether a sequence of actions can be executed or to determine the outputs produced in response to a sequence of inputs. Equivalence queries on the other hand correspond to the question whether a hypothesis model produced by the learner represents the SUL. The teacher either answers affirmatively or with a counterexample showing non-equivalence between the SUL and the hypothesis.

✉ Bernhard K. Aichernig
aichernig@ist.tugraz.at

1 Institute of Software Technology, Graz University of Technology, Graz, Austria

The first type of query is simple to implement for learning black-box systems. It generally suffices to reset the system, execute a single test and record observations. Equivalence queries, however, are more difficult to implement. Peled et al. [26], as one of the first to combine learning and formal verification, proposed to implement these queries via conformance testing. In particular, they suggested to use the conformance testing algorithm by Vasilevskii [35] and Chow [9].

This method is also referred to as W-method and there exist optimisations of it, like the partial W-method [16] or an approach by Lee and Yannankakis [22], but all have the same worst-case complexity [7]. All three methods share two issues. They require a fixed upper bound on the number of states of the black-box system which is generally unknown. Additionally, the size of the constructed test suite is exponential in this bound. Therefore, implementing the equivalence oracle can be considered “the true bottleneck of automata learning” [7].

In practice, there is limited time for testing and therefore also for learning. The ZULU challenge [10] addressed this issue by limiting the number of tests to be executed [17]. More concretely, competitors learned finite automata from a limited number of membership queries without explicit equivalence queries. Equivalence queries thus had to be approximated through clever selection of membership queries. This led to a different view of the problem: rather than “trying to prove equivalence”, the new target was “finding counterexamples fast” [17].

We propose an implementation of equivalence queries based on mutation testing [20], more specifically on model-based mutation testing [2]. This approach follows the spirit of the ZULU challenge by trying to minimise the number of tests for executing equivalence queries. We use a combination of random testing, to achieve high variability of tests, and mutation analysis, to address coverage appropriately. To illustrate the effectiveness of our approach, which has been implemented based on the LearnLib library [19], we will mainly compare it to the partial W-method [16] and show that the cost of testing can be significantly reduced while still learning correctly. In other words, our method reliably finds counterexamples with less testing. In addition to that, we also compare it to purely random testing and to an effective implementation of a randomised conformance testing method described by Smeenk et al. [30].

Parts of this paper have been published in the proceedings of the 9th NASA Formal Methods Symposium (NFM 2017) [4]. In this extended version, we provide an explicit and thorough description of our mutation technique and of our optimised mutation analysis. Additionally, we significantly enhanced the evaluation. It now covers experiments with models of 16 systems. Originally, we presented experiments with models of only two systems.

We target systems which can be modelled with a moderately large number of states, i.e. with up to about fifty states. This restriction is necessary, because mutation analysis is a computationally intensive task for large systems. Nevertheless, there exists a wealth of non-trivial systems, such as implementations of communication protocols, which can be learned nonetheless. This is feasible if suitable abstractions are applied, which is generally done in learning-based verification.

The rest of this paper is structured as follows. Section 2 discusses related work and Sect. 3 introduces preliminaries along with an example demonstrating active automata learning. The main part of this article comprises three sections. Section 4 presents our proposed process for test-suite generation involving mutation-coverage-based selection, while Sect. 5 introduces a mutation technique tailored towards active automata learning. The evaluation of the test-suite generation is shown in Sect. 6, which is based on our implementation available at [32]. We conclude the paper in Sect. 7.

2 Related Work

We address conformance testing in active automata learning. Hence, there is a relationship to the W-method [9,35] and the partial W-method [16], two conformance testing methods implemented in LearnLib [19]. However, we handle fault coverage differently. By generating tests to achieve transition coverage, we also test for "output" faults, but do not check for "transfer" faults. Instead we present a fault model directly related to the specifics of learning in Sect. 5.

We combine model-based mutation testing and random testing, which we discussed in previous work [1,2]. Generally, random testing is able to detect a large number of mutants fast, such that only a few subtle mutants need to be checked with directed search techniques. While we do not aim at detecting all mutants, i.e. we do not apply directed search, this property provides a certain level of confidence. By analysing mutation coverage of random tests, we can guarantee that detected mutations do not affect the learned model.

Howar et al. [17] noted that it is necessary to find counterexamples with few tests for automata learning to be practically applicable. We generally follow this approach. Furthermore, one of the heuristics described in [17] is based on Rivest and Schapire's counterexample processing [28], similar to the fault model discussed in Sect. 5. More recent work in this area has been performed by Smeenk et al. [30], who implemented a partly randomised conformance testing technique. In order to keep the number of tests small, they applied a technique to determine an adaptive distinguishing sequence described by Lee and Yannakakis [21]. With this technique and domain-specific knowledge, they succeeded in learning a large model of industrial control software. The same technique has also been used to learn models of Transmission Control Protocol (TCP) implementations [14] and of SSH implementations [15].

3 Preliminaries

3.1 Mealy Machines

We use Mealy machines because they are well-suited to model reactive systems and they have successfully been used in contexts combining learning and some form of verification [11,14, 23,33]. Moreover, the Java-library LearnLib [19] 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. Additionally, exactly one output is produced in response to each input. Formally, Mealy machines can be defined as follows.

Definition 1 A Mealy machine $\mathcal{M}$ is a 6-tuple $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ where

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

We require Mealy machines to be input-enabled and deterministic. The former demands that outputs and successor states must be defined for all inputs and all states, i.e. δ and λ must be total. A Mealy machine is deterministic if it defines at most one output and successor state for every pair of input and state, i.e. δ and λ must be functions in the mathematical sense.

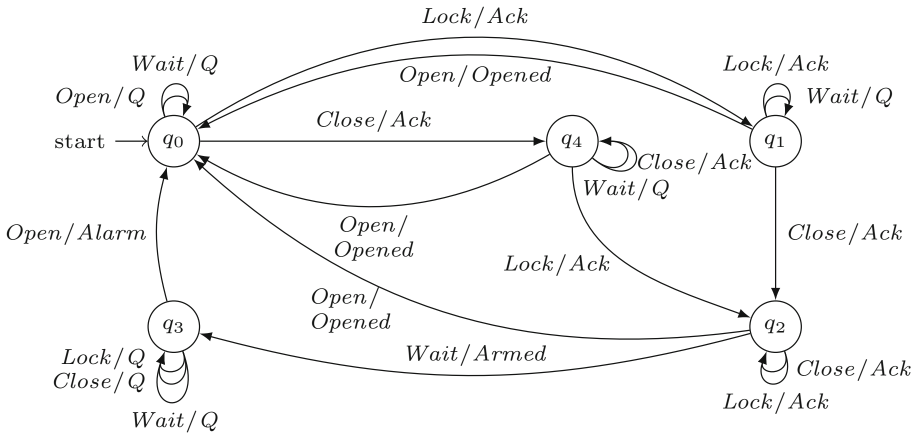


Fig. 1 A Mealy machine modelling a car alarm system

Example 1 (*Mealy Machine Model of a Car Alarm System*) Figure 1 shows a Mealy machine modelling a simple car alarm system. It has the inputs *Open*, *Close*, and *Lock* for opening, closing, and locking the doors of a car. Another input *Wait* denotes waiting for some time. Depending on the current state, the system reacts with one of the outputs $O = \{Opened, Ack, Q, Armed, Alarm\}$ to each input, where Q denotes quiescent behaviour, i.e. the absence of outputs. There are two main requirements for the car alarm system: (1) it must be *Armed* if the doors have been closed and locked for some time and (2) it must produce an *Alarm* if the doors are opened in the *Armed* state.

Transitions of the Mealy machine are labelled by pairs of inputs and outputs separated by a slash. We have for instance $\delta(q_2, Wait) = q_3$ and $\lambda(q_2, Wait) = Armed$.

Notational Conventions and Terminology Let $s, s' \in S^*$ be two sequences of input/output symbols, i.e. $S = I$ or $S = O$, then $s \cdot s'$ denotes the concatenation of these sequences. The empty sequence is represented by ϵ. The length of a sequence is given by $|s|$. We implicitly lift single elements to sequences, thus for $e \in S$ we have $e \in S^*$ with $|e| = 1$. As a result, the concatenation $s \cdot e$ is also defined. Zero-based indexed access to the ith element of a sequence $s = s_0 \cdots s_{n-1}$ is denoted by $s[i] = s_i$. Expressions of the form $s[\oplus i]$, where $\oplus \in \{<, \leq, >, \geq\}$, may be used to select prefixes/suffixes of s consisting of elements with indexes j satisfying $j \oplus i$, e.g. $s[< i] = s_0 \cdots s_{i-1}$. A set of sequences S is prefix-closed, if it contains all prefixes of each element of S.

We extend δ and λ to sequences of inputs in the standard way. Let $s \in I^*$ be an input sequence and $q \in Q$ be a state, then $\delta(q, s) = q' \in Q$ is the state reached by executing s starting in state q. For $s \in I^*$ and $q \in Q$, the output function $\lambda(q, s) = t \in O^*$ returns the outputs produced in response to s executed in state q. Furthermore, let $\lambda(s) = \lambda(q_0, s)$ and $\delta(s) = \delta(q_0, s)$. For a state q, the set $acc(q) = \{s \in I^* | \delta(q_0, s) = q\}$ contains the access sequences of q, i.e. the sequences leading to q. Note that other authors define a unique access sequence $s \in I^*$ for each q [18].

In learning, we need to determine whether two Mealy machines are equivalent. Equivalence is usually defined with respect to outputs [14], i.e. two deterministic Mealy machines are equivalent if they produce the same outputs for all input sequences. Two Mealy machines $\langle Q_1, q_{01}, I, O, \delta_1, \lambda_1\rangle$ and $\langle Q_2, q_{02}, I, O, \delta_2, \lambda_2\rangle$ are equivalent 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 $\lambda_1(q_{0_1}, s) \neq \lambda_2(q_{0_2}, s)$.

3.2 Active Automata Learning

We consider learning in the minimally adequate teacher (MAT) framework [5]. Algorithms in this framework infer models of black-box systems, also referred to as SULs, through interaction with a so-called teacher.

3.2.1 Minimally Adequate Teacher Framework

The interaction is carried out via two types of queries posed by the learning algorithm and answered by a MAT. 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 is used to learn a deterministic finite automaton (DFA) representing a regular language known to the teacher [5]. Given some alphabet, the L^* algorithm repeatedly selects strings and asks membership queries 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.

After processing a counterexample, learning algorithms 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 as described in the following.

3.2.2 Learning Mealy Machines

Margaria et al. [23] and Niese [25] were among the first to infer Mealy-machine models of reactive systems using an L^*-based algorithm. Another L^*-based learning algorithm for Mealy machines has been presented by Shahbaz and Groz [29]. They reuse the structure of L^*, but substitute membership queries for *output queries*. Instead of checking whether a string is accepted, they provide inputs and the teacher responds with the corresponding outputs. For a more practical discussion, consider the instantiation of a teacher. Usually we want to learn the behaviour of a black-box SUL of which we only know the interface. Hence, output queries are conceptually simple: provide inputs to the SUL and observe produced outputs. However, there is a slight difficulty hidden. Shahbaz and Groz [29] assume that outputs are produced in response to inputs executed from the **initial** state. Consequently, we need to have some means to reset a system. As discussed in the introduction, we generally cannot check for equivalence. It is thus necessary to approximate equivalence queries, e.g., via

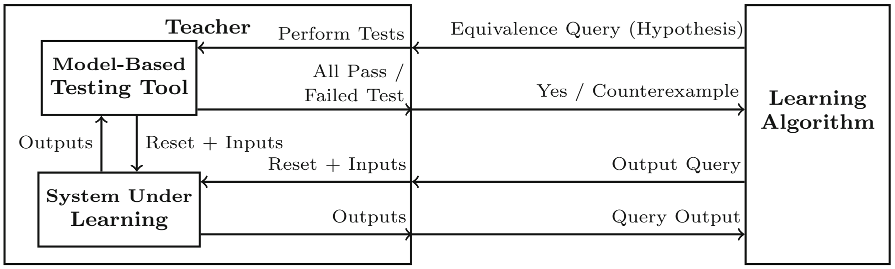


Fig. 2 The interaction between SUL, teacher and learning algorithm [30]

Algorithm 1 General pattern for active automata learning from black-box SULs.

```
Input: SUL with a RESET operation and a function TEST : $I^* \rightarrow O^*$ executing a sequence of inputs and
    collecting the produced outputs
Output: learned Mealy machine
 1: repeat
 2:     $t \leftarrow$ select sequence for output query
 3:     RESET
 4:     $u \leftarrow$ TEST($t$)                                        ▷ perform output query
 5:     store $(t, u)$
 6: until sufficient information to build hypothesis
 7: $\mathcal{H} = \langle Q, q_0, I, O, \delta, \lambda \rangle \leftarrow$ build hypothesis
 8: Generate test suite $T \subset I^*$ from $\mathcal{H}$                ▷ perform equivalence query
 9: for all $t \in T$ do
10:     RESET
11:     $u \leftarrow$ TEST($t$)
12:     if $u \neq \lambda(q_0, t)$ then
13:         extract information from counterexample $t$ and goto 1
14:     end if
15: end for
16: output $\mathcal{H}$
```

conformance testing as implemented in LearnLib [19]. To summarise, a learning algorithm for Mealy machines relies on three operations:

1. *reset:* resets the SUL
2. *output query:* performs a single test executing inputs and recording outputs
3. *equivalence query:* conformance testing between SUL and hypothesis

As shown in Fig. 2, the teacher is usually a component communicating with the SUL. An equivalence query results in a positive answer if all conformance tests pass, i.e. the SUL produces the same outputs as the hypothesis. If there is a failing test, the corresponding input sequence is returned as counterexample.

Learning Process for Black-Box Systems Based on the three operations described above, active automata learning of black-box SULs usually implements the general pattern given in pseudo-code by Algorithm 1. We can see that this algorithm follows the process introduced for L^*. It asks output queries until it can build a hypothesis (Line 1 to Line 6). Then, it asks an equivalence query via testing (Line 8 to Line 15), and if it finds a counterexample to equivalence, it goes back to asking output queries. Aside from this general process, there are also algorithm-dependent operations, which we left abstract, like how output queries are selected (Line 2) or how information is stored (Line 5).

Table 1 Initial observation table for the car alarm system

S	E			
	Open	Lock	Close	Wait
S				
ϵ	Q	Ack	Ack	Q
$S{\cdot}I$				
Lock	Opened	Ack	Ack	Q
Open	Q	Ack	Ack	Q
Close	Opened	Ack	Ack	Q
Wait	Q	Ack	Ack	Q

We will now discuss relevant aspects of the learning process on the basis of the car alarm system introduced in Example 1, by performing one round of learning, i.e. executing Algorithm 1 until Line 13.

The way how information is stored, what output queries are performed, and how much information is necessary to build a hypothesis depends on the specific learning algorithm. Common to various approaches is that they maintain a set of input sequences identifying states of the hypothesis [5,18,31]. The states reached by these sequences are usually distinguished by outputs produced in response to another set of input sequences. In other words, states are distinguished by their future behaviour [31]. We denote the first set of sequences by S and the distinguishing input sequences by E. In L^*-based learning, we need to perform output queries for all $s \cdot e$, with $s \in S$ and $e \in E$, therefore we refer to S as prefixes and to E as suffixes.

Example 2 (*Deriving a First Hypothesis with*L^*_M) The information gained during learning via L^*_M is stored in an observation table [5,31], whose structure is defined by a prefix-closed set of prefixes $S \subset I^*$ identifying states and a set of suffixes $E \subset I^*$. Commonly these sets are initialised to $S = \{\epsilon\}$ and $E = I$. The rows of the table are labelled by elements S and $S \cdot I$, i.e. their one-input-extensions, while the columns are labelled by elements from E. To fill the table cells, we perform output queries for $S \cdot E$ and $S \cdot I \cdot E$ and store outputs of the SUL in response to suffixes from E (note that this is sufficient, because S is prefix-closed). The initial observation table for the car alarm system is given by Table 1. This table however is not closed, therefore we cannot build a hypothesis (condition in Line 6 of Algorithm 1 is not satisfied). Consequently, we add further elements to S and E while performing additional output queries until the observation table is closed and consistent [5]. Since these aspects are not relevant to the scope of the paper, we refer to [5] for a formal definition of closedness and consistency. In our example, we add *Lock*, *Lock* $\cdot$ *Close*, and *Lock* $\cdot$ *Close* $\cdot$ *Wait* to S, such that we can build the hypothesis shown in Fig. 3. Note that S now contains access sequences of all four states of the hypothesis, the sequences in E distinguishes them, and the one-input-extensions $S \cdot I$ define transitions in the hypothesis.

In Line 8 of Algorithm 1, we need to derive a set of test cases to perform an equivalence query. Observe the similarity to the definition of equivalence between Mealy machines, which requires equivalent outputs in response to all sequences of inputs. Since we cannot test all input sequences, i.e. due to the incompleteness of testing, learned models may be incorrect. If we, e.g., test with the W-method [9,35], the learned model may be incorrect if the assumed maximum number of states of the SUL is too low. This suggests that we should generate meaningful tests with low redundancy to cover as much of the potential state space

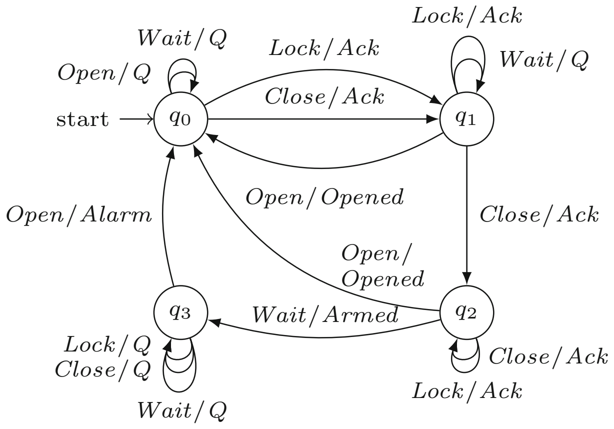


Fig. 3 First hypothesis derived during learning

of the black-box SUL as possible. This is the main goal of the test-case generation approach presented in this paper. Another possible conclusion from the incompleteness of testing is that testing should concentrate on finding counterexamples rather than showing equivalence as suggested by Howar et al. [17]. In the following example, we present a counterexample between the SUL in Fig. 1 and the hypothesis shown in Fig. 3. We further demonstrate a way to extract relevant information from the counterexample (Line 13 of Algorithm 1). Note that there are various alternative ways to process counterexamples.

Example 3 (*A Counterexample for the Car Alarm System*) Although the first hypothesis in Fig. 3 has only four states, executing the partial W-method [16] with a depth of 2, i.e. assuming the SUL has at most 6 states, generates already 546 tests. One of these tests is the input sequence $t = Close \cdot Lock \cdot Wait$. The hypothesis produces the outputs $Ack \cdot Ack \cdot Q$ for t, while the SUL produces $Ack \cdot Ack \cdot Armed$ (see Fig. 1). Intuitively, this counterexample reveals the fault that *Close* and *Lock* lead to the same state, if performed in the initial state q_0.

We can extract this information from t via a decomposition of t presented by Rivest and Schapire [28]. Actually, they introduced this decomposition in combination with "reduced observation tables" [28,31], which imposes constraints on how observation tables are extended. They basically required all sequences in S to be distinguished by sequences in E. The rationale behind the decomposition is that the existence of a counterexample implies that there are two sequences s, s' in S and $S \cdot I$ leading to the same state, although they reach different states in the SUL. The decomposition extracts a distinguishing suffix v showing that the future behaviours of s and s' are different, i.e. the sequences should lead to different states. Adding v to set E causes at least one new state to be added to the hypothesis derived in the next round of learning. Here, in this example we extract $Lock \cdot Wait$ from t which shows that *Close* and *Lock* reach different states in the SUL. As a result, the next round of learning adds the state q_4 resulting the final model shown in Fig. 1.

We built our mutation technique in relation to the counterexample decomposition technique by Rivest and Schapire [28], which we will present in Sect. 5 along with further details on the actual decomposition. The general idea behind our test-case generation approach is to generate test cases, which are able to distinguish the current hypothesis from as many potential succeeding hypotheses as possible.

Algorithm 2 The test-case generation algorithm.

```
Input: $\mathcal{M}_h = \langle S_h, s_{0h}, I, O, \lambda_h, \delta_h \rangle$, $p_{retry}$, $p_{stop}$, $maxSteps$, $l_{infix}$,
Output: $test$
1: $state \leftarrow s_{0h}$
2: $test \leftarrow \epsilon$
3: if $coinFlip(0.5)$ then
4:     $test \leftarrow rSeq(I, l_{infix})$
5:     $state \leftarrow \delta(state, test)$
6: end if
7: loop
8:     $rS \leftarrow rSel(S_h)$                ▷ $(rS, rI)$ defines
9:     $rI \leftarrow rSel(I)$                  ▷ a transition
10:    $p \leftarrow path(state, rS)$
11:    if $p \neq None$ then
12:        $rSteps \leftarrow rSeq(I, l_{infix})$
13:        $test \leftarrow test \cdot p \cdot rI \cdot rSteps$
14:        $state \leftarrow \delta(\delta(rS, rI), rSteps)$
15:        if $|test| > maxSteps$ then
16:            break
17:        else if $coinFlip(p_{stop})$ then
18:            break
19:        end if
20:    else if $\neg coinFlip(p_{retry})$ then
21:        break
22:    end if
23: end loop
```

4 Test-Suite Generation

We had shown previously "that by adding mutation testing to a random testing strategy approximately the same number of bugs were found with fewer test cases" [2]. Motivated by this, we developed a simple, yet effective test-suite generation technique. The test-suite generation has two parts, (1) generating a large set of tests T and (2) selecting a subset $T_{sel} \subset T$ to be run on the SUL.

4.1 Test-Case Generation

The goal of the test-case generation is to achieve high coverage of the model under consideration combined with variability through random testing. Algorithm 2 implements this form of test-case generation. The procedure may start with a random walk through the model (Line 3 to Line 6) and then iterates two operations. First, a transition of the model is chosen randomly and a path leading to it is executed (Line 8 to Line 10). If the transition is not reachable, another target transition is chosen. Second, another short random walk is executed (Line 12). These two operations are repeated until a stopping criterion is reached.

Stopping Test-case generation stops as soon as the test has a length greater than a maximum number of steps *maxSteps* (Line 15). Alternatively, it may also stop dependent on probabilities p_{retry} (Line 20) and p_{stop} (Line 17). The first one controls the probability of continuing in case a selected transition is not reachable while the second one controls the probability of stopping prematurely.

Random Functions Test generation uses three random functions. The function *coinFlip* is defined for $p \in [0, 1]$ by $\mathbb{P}(coinFlip(p) = true) = p$ and $\mathbb{P}(coinFlip(p) = false) = 1 - p$. The function *rSel* selects a single sample from a set according to a uniform distribution, i.e. $\forall e \in S\colon \mathbb{P}(rSel(S) = e) = \frac{1}{|S|}$. The function *rSeq* takes a set S and a bound $b \in \mathbb{N}$ and creates a sequence of length $l \leq b$ consisting of elements from S chosen via *rSel*, whereby l is chosen uniformly from $[0\,..\,b]$.

We assume a given Mealy machine $\mathcal{M}_h = \langle S_h, s_{0h}, I, O, \lambda_h, \delta_h \rangle$ in the following. This Mealy machine is also one of the parameters of Algorithm 2. The test-case generation is further controlled by the stopping parameters and $l_{infix} \in \mathbb{N}$, an upper bound on the number

Algorithm 3 Breadth-first exploration implementing *path*.

```
Input: $\mathcal{M}_h = \langle S_h, s_{0h}, I, O, \lambda_h, \delta_h \rangle$, arguments $s, s'$ of $path$
Output: either $path \in I^*$ such that $\delta(s, path) = s'$ or $None$
  $visited \leftarrow \{\}$
  $next \leftarrow emptyQ()$                              ▷ an empty queue of states to explore
  $next \leftarrow enqueue(next, (s, \epsilon))$            ▷ and traces leading to those states
  while $next \neq emptyQ()$ do
    $(s_c, path) \leftarrow dequeue(next)$
    if $s_c \notin visited$ then                          ▷ current state not visited yet
      $visited \leftarrow visited \cup \{s_c\}$
      for all $i \in I$ do
        $s_n \leftarrow \delta(s_c, i)$
        if $s_n = s'$ then                                ▷ we found the target state
          return $path \cdot i$
        end if
        $next \leftarrow enqueue(next, (s_n, path \cdot i))$
      end for
    end if
  end while
  return $None$                                           ▷ did not find a path to the target $s'$
```

of random steps executed between visiting two transitions. The function *path* returns a path leading from the current state to another state. Currently, this is implemented via a breadth-first exploration given by Algorithm 3, but other approaches are possible as long as they satisfy $path(s, s') = None$ iff $\nexists \bar{i} \in I^* : \delta(s, \bar{i}) = s'$ and $path(s, s') = \bar{i} \in I^*$ such that $\delta(s, \bar{i}) = s'$, where $None \notin I$ denotes that no such path exists.

4.2 Test-Case Selection

To account for variations in the quality of randomly generated tests, not all generated tests are executed on the SUL, but rather a selected subset. This selection is based on coverage, e.g., transition coverage. For the following discussion, assume that a set of tests T_{sel} of fixed size n_{sel} should be selected from a previously generated set T to cover elements from a set C. In a simple case, C can be instantiated to the set of all transitions, i.e. $C = S_h \times I$ as $(s, i) \in S_h \times I$ uniquely identifies a transition because of determinism. The selection comprises the following steps:

1. The coverage of single test cases is analysed, i.e. each test case $t \in T$ is associated with a set $C_t \subseteq C$ covered by t.
2. The actual selection has the objective of optimising the overall coverage of C. It is implemented by Algorithm 4 . We greedily select test cases until either the upper bound n_{sel} is reached (Line 2 and Line 3), all elements in C are covered (second condition in Line 2), or we do not improve coverage (Line 4 and Line 5).
3. If n_{sel} tests have not yet been selected, then further tests are selected which individually achieve high coverage. For that $t \in T \setminus T_{\text{sel}}$ are sorted in descending size of C_t and the first $n_{\text{sel}} - |T_{\text{sel}}|$ tests are selected.[1]

[1] Note that more sophisticated test suite reduction/prioritisation strategies could be used. However, this is beyond the scope of this paper.

Algorithm 4 Coverage-based test selection

```
Input: $T, C, C_t$ for all $t \in T$, $n_{\text{sel}}$
Output: $T_{\text{sel}}$
1: $T_{\text{sel}} \leftarrow \emptyset$
2: while $|T_{\text{sel}}| < n_{\text{sel}} \wedge C \neq \emptyset$ do
3:     $t_{\text{opt}} \leftarrow \text{argmin}_{t \in T} |C \setminus C_t|$
4:     if $C \cap C_{t_{\text{opt}}} = \emptyset$ then
5:         break                                   ▷ no improvement
6:     end if
7:     $T_{\text{sel}} \leftarrow T_{\text{sel}} \cup \{t_{\text{opt}}\}$
8:     $C \leftarrow C \setminus C_{t_{\text{opt}}}$
9: end while
```

4.3 Mutation-Based Selection

A particularly interesting selection criterion is selection based on mutation. The choice of this criterion is motivated by the fact that model-based mutation testing can effectively be combined with random testing [2]. Generally, in this fault-based test-case generation technique, known faults are injected into a model creating so-called mutants. Test cases are generated which distinguish these mutants from the original model and thereby test for the corresponding faults.

Thus, in our case we alter the hypothesis $\mathcal{M}_{\text{h}}$, creating a set of mutants $\mathcal{MS}_{\text{mut}}$. The objective is now to distinguish mutants from the hypothesis, i.e. we want tests that show that mutants are observably different from the hypothesis. Hence, we can set $C = \mathcal{MS}_{\text{mut}}$ and $C_t = \{\mathcal{M}_{\text{mut}} \in \mathcal{MS}_{\text{mut}} \mid \lambda_{\text{h}}(t) \neq \lambda_{\text{mut}}(t)\}$.

4.3.1 Type of Mutation

The type of faults injected into a model is governed by mutation operators, which basically map a model to a set of mutated models (mutants). There is a variety of operators for programs [20] and also finite-state machines [13]. As an example, consider a mutation operator *change output* which changes the output of each transition and thereby creates one mutant per transition. Since there is exactly one mutant that can be detected by executing each transition, selection based on such mutants is equivalent to selection with respect to transition coverage. Hence, mutation can simulate other coverage criteria. In fact, for our evaluation we implemented transition coverage via mutation.

Blindly using all available mutation operators may not be effective. Fault-based testing should rather target faults likely to occur in the considered application domain [27]. Thus, we developed a family of mutation operators, called *split-state* operators, directly addressing active automata learning. We will discuss this kind of mutation in Sect. 5.

4.4 The Complete Process

Now that we have discussed test-case generation, coverage-based selection, and mutation-based selection in particular, we want to give an overview of the proposed process for executing an equivalence query *eq* via mutation-based conformance testing. The data flow of the complete process involving test suite derivation and execution is shown in Fig. 4.

The input for this process is a hypothesis $\mathcal{H}$, a learned intermediate Mealy machine model. From this model $\mathcal{H}$, we generate a set T of randomised test cases via Algorithm 2 and we generate a set $\mathcal{MS}_{\text{mut}}$ of mutants. Then, we analyse the mutation coverage C_t of each test case $t \in T$; i.e. we execute each t and determine which of the mutants produces outputs different from the hypothesis' outputs. Next, the test suite for conformance testing is

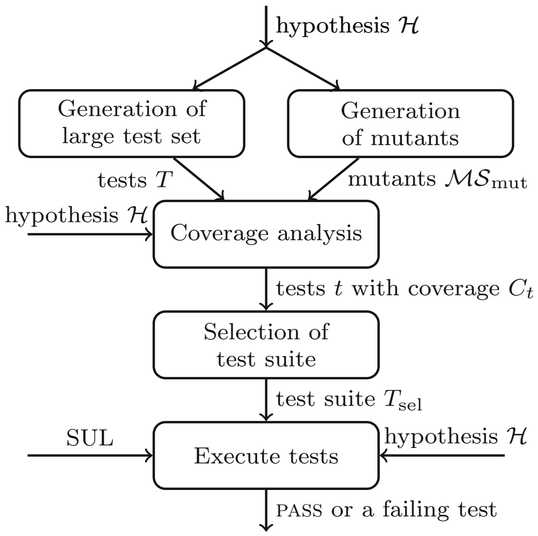


Fig. 4 Data flow for test-suite generation and execution

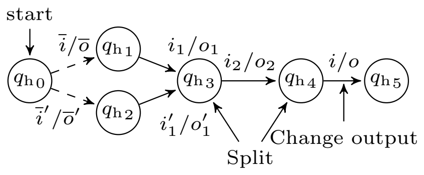
created by selecting a subset T_{sel} of T based on the coverage information C_t and by applying Algorithm 4. Finally, we execute all test cases of the test suite T_{sel}. A test case producing different outputs on the SUL than the outputs predicted by the hypothesis is a counterexample to equivalence. If such a counterexample is found, it is returned to the learning algorithm. If no counterexample is found, we assign a PASS verdict to the test suite. Such a PASS verdict translates to a positive answer to the equivalence query eq.

5 Mutation for Learning

We gave a general overview of test-suite generation in the previous section. In the following, we present a mutation technique specifically targeting conformance testing in active automata learning. Hence, instantiating the process shown in Fig. 4 with this technique yields an efficient testing method for learning. In the following, we will first introduce *split state* mutation operators, defining the form of mutants in our implementation of mutation testing. After that, we will present our implementation of mutant generation and optimised mutation analysis. We conclude the section with specifics affecting mutant generation and mutation analysis.

5.1 Split-State Mutation Operator Family

There are different ways to process counterexamples in the MAT framework, such as by adding all prefixes to the used data structures [5]. An alternative technique due to Rivest and Schapire [28] takes the "distinguishing power" of a counterexample into account. The basic idea is to decompose a counterexample into a prefix u, a single action a and a suffix v such that v is able to distinguish access sequences in the current hypothesis. In other words, the distinguishing sequence v shows that two access sequences, which were hypothesised to lead to the same state, actually lead to observably nonequivalent states (see also Example 3). This knowledge is then integrated into the data structures.



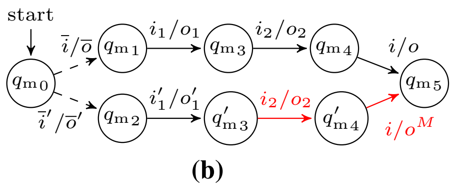


Fig. 5 Demonstration of split state. **a** A possible hypothesis model, **b** A possible SUL/mutant

Since it is an efficient form of processing counterexamples, adaptations of it have been used in other learning algorithms such as the TTT algorithm [18]. This algorithm makes the effect of this decomposition explicit. It *splits* a state q reached by an access sequence derived from u and a. The splitting further involves (1) adding a new state q' reached by another access sequence derived from u and a (which originally led to q) and (2) adding sequences to the internal data structures which can distinguish q and q'.

The development of the *split-state* family of mutation operators is motivated by the principle underlying the TTT and related algorithms. Basically, we collect pairs (u, u') of access sequences of a state q, add a new state q' and redirect u' to q'. Furthermore, we add transitions such that q' behaves the same as q except for a distinguishing sequence v. Example 4 illustrates this mutation operator.

Example 4 (*Split State Mutation*) A hypothesis produced by a learning algorithm may be of the form shown in Fig. 5a. Note that not all edges are shown in the figure and dashed edges represent sequences of transitions. The access sequences $acc(q_{h3})$ of q_{h3} thus include $\bar{i} \cdot i_1$ and $\bar{i}' \cdot i'_1$. A possible corresponding black-box SUL is shown in Fig.5b. In this case, the hypothesis incorrectly assumes that $\bar{i} \cdot i_1$ and $\bar{i}' \cdot i'_1$ lead to the same state. We can model a transformation from the hypothesis to the SUL by splitting q_{h3} and q_{h4} and changing the output produced in the new state $q_{m'_4}$ as indicated in Fig. 5b. State q_{h4} has to be split as well to introduce a distinguishing sequence of length two while still maintaining determinism. A test case covering the mutation is $\bar{i}' \cdot i'_1 \cdot i_2 \cdot i$.

A mutant models a SUL containing two different states q and q' which are assumed to be equivalent by the hypothesis. By executing a test covering a mutant $\mathcal{M}_{\text{mut}}$, we either find an actual counterexample to equivalence between SUL and hypothesis or prove that the SUL does not implement $\mathcal{M}_{\text{mut}}$. Hence, it is possible to guarantee that the SUL possesses certain properties. This is similar to model-based mutation testing in general, where the absence of certain faults, those modelled by mutation operators, can be guaranteed [2].

Split state is a family of mutation operators as the effectiveness of the approach is influenced by several parameters, such that the instantiation of parameters can be considered a unique operator. The parameters are:

1. *Max. number of sequences* n_{acc}: an upper bound on the number of mutated access sequences leading to a single state.
2. *Length of distinguishing sequences* k: for each splitting operation we create $|I|^k$ mutants, one for each sequence of length k. Note that this requires the creation of k new states. Coverage of all mutants generated with length k implies coverage of all mutants with length $l < k$.
3. *Split at prefix flag:* redirecting a sequence $u' \cdot a$ from q to q' usually amounts to changing $\delta(\delta(s_{0h}, u'), a) = q$ to $\delta(\delta(s_{0h}, u'), a) = q'$. However, if the other access sequence in

the pair is $u \cdot a$ with $\delta(s_{0\mathrm{h}}, u') = \delta(s_{0\mathrm{h}}, u)$, this is not possible because it would introduce non-determinism. This flag specifies whether the access sequence pair $(u \cdot a, u' \cdot a)$ is ignored or whether further states are added to enable redirecting $u' \cdot a$. We generally set it to *true*.

5.2 Implementation of Mutant Generation

The generation of mutants is implemented by Algorithm 5. It requires the auxiliary function $subset(S, k) = S'$ where $|S'| = k$, $S' \subset S$ if $|S| > k$ and $S' = S$ otherwise. In the main loop (Line 2 to Line 6), we look at all unordered pairs $\{s_1, s_2\}$ of n_{acc} access sequences for a state q. Via the function SPLIT, we create mutants for these pairs. This function returns the empty set if one sequence is a prefix of the other (Line 8). The reason for this is that we would alter the behaviour for an input sequence v if we would redirect a prefix of v.

Otherwise, we compute the suffix *eqSuf* along which both s_1 and s_2 visit the same states and execute the same inputs (Line 10). Then, we decompose s_1 into a prefix s_1', an input *pI*, and the suffix *eqSuf*. Mutants are created for all distinguishing sequences *distSeq* (Line 13 to Line 41). Initially, mutants are copies of the original Mealy machine except for the state q_{sp} reached by $s_1' \cdot pI$, the state which is split, and the transition leading to this state (Line 14 to Line 16). We create a new state q_s for q_{sp} reached by *pI*. Furthermore, we create states along the sequence *eqSuf* (see *Split at prefix flag* above) to ensure determinism and along the distinguishing sequence *distSeq* (Line 27 to Line 31). Finally, we mutate the last output corresponding to the last input of *distSeq* (Line 25). All other transitions, added in lines 27 to 31 and in lines 35 to 38, ensure that the mutant produces the same outputs as the original hypothesis if the distinguishing sequence is not executed.

In the implementation of $acc(q)$, we collect access sequences by performing a breadth-first exploration while traversing every state at most twice. Therefore, we may not find all n_{acc} access sequences for state q. This strategy is motivated by performance considerations and the mutant sampling strategy *reduce to min*, which we commonly use.

5.3 Efficiency Considerations and Optimisation

While test-case generation can efficiently be implemented, mutation-based selection is generally computationally intensive. It is necessary to check which of the mutants is covered by each test case. Since the number of mutants may be as large as $|S| * n_{\mathrm{acc}} * |I|^k$, this may become a bottleneck. Consequently, cost reduction techniques for mutation [20] need to be considered.

5.3.1 Optimisation of Mutation Analysis

We reduce execution cost by avoiding the explicit creation of mutants. Essentially only the difference to the hypothesis is stored and executed. This optimisation is based on the following observation. In Algorithm 5, we can see that a mutant is uniquely identified by the 3-tuple $\langle \mathcal{M}, (q_{\mathrm{pre}}, pI), v\rangle$ where $\mathcal{M}$ is the original Mealy machine, (q_{pre}, pI) is a transition of $\mathcal{M}$ (Line 14) and $v = eqSuf \cdot distSeq$ is the sequence leading to the mutated output (Line 18). Hence, we basically check for coverage of a combination consisting of: (1) a sequence leading to transition (q_{pre}, pI), (2) the transition (q_{pre}, pI), and (3) the sequence v.

Given that insight, we implemented an efficient mutation analysis technique. Instead of explicitly creating all mutants, we rather generate all 3-tuples which implicitly describe all

Algorithm 5 The mutant generation algorithm

```
Input: hypothesis $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda\rangle$,$n_{acc}$, $k$
Output: set of mutants $Mut$
1: $Mut \leftarrow \{\}$
2: for $q \in Q$ do                         $\triangleright$ mutants for all unordered pairs of access sequences of $q$
3:     for $\{s_1, s_2\} \in \{\{s_1, s_2\} | s_1, s_2 \in subset(acc(q), n_{acc})\}$ do
4:         $Mut \leftarrow Mut \cup \textsc{Split}(s_1, s_2, \mathcal{M}, k) \cup \textsc{Split}(s_2, s_1, \mathcal{M}, k)$
5:     end for
6: end for

7: function $\textsc{Split}(s_1, s_2, \langle Q, q_0, I, O, \delta, \lambda\rangle, k)$
8:     if $\exists s : s_1 \cdot s = s_2$ then return $\{\}$                    $\triangleright$ $s_1$ prefix of $s_2$ ?
9:     else
10:        $eqSuf \leftarrow \text{argmax}_s \{|s| \mid \exists s'_1, s'_2 : s'_1 \cdot s = s_1 \wedge s'_2 \cdot s = s_2 \wedge \delta(s'_1) = \delta(s'_2)\}$
11:        $s'_1 \cdot pI \cdot eqSuf \leftarrow s_1$          $\triangleright$ equivalent behaviour of $s_1$ and $s_2$ along $eqSuf$
12:        $Mutants \leftarrow \{\}$
13:        for all $distSeq \in I^k$ do          $\triangleright$ all possible distinguishing sequences of length $k$
14:            $q_{sp} \leftarrow \delta(s'_1 \cdot pI)$, $q_{pre} \leftarrow \delta(s'_1)$                    $\triangleright$ $q_{sp}$ is split
15:            $\mathcal{M}' \leftarrow \langle Q', q_0, I, O, \delta', \lambda'\rangle$ s.t. $Q' = Q, \lambda' = \lambda$, and $\delta' = \delta \setminus \{(q_{pre}, pI, q_{sp})\}$
16:            $Q' \leftarrow Q' \cup \{q_s\}$, $\delta' \leftarrow \delta' \cup \{(q_{pre}, pI, q_s)\}$ for a new $q_s \notin Q'$          $\triangleright$ copy original
17:            $q_{pre_{mut}} \leftarrow q_s$
18:            $v \leftarrow eqSuf \cdot distSeq$
19:            $newQ \leftarrow \{\}$
20:            for $j = 0$ to $|v| - 1$ do
21:                $q_{orig} \leftarrow \delta(s'_1 \cdot pI \cdot v[< j])$
22:                $q_{n_{orig}} \leftarrow \delta(s'_1 \cdot pI \cdot v[\leq j])$
23:                if $j = |v| - 1$ then          $\triangleright$ mutation transition back to original states
24:                    $\delta' \leftarrow \delta' \cup \{(q_{pre_{mut}}, v[j], q_{n_{orig}})\}$
25:                    $\lambda' \leftarrow \lambda' \cup \{(q_{pre_{mut}}, v[j], o_{mut})\}$ s.t. $o_{mut} \neq \lambda(q_{orig}, v[j])$
26:                else                         $\triangleright$ new states with indistinguishable behaviour
27:                    $Q' \leftarrow Q' \cup \{q_n\}$, for a new $q_n \notin Q'$
28:                    $newQ \leftarrow newQ \cup \{(q_n, q_{n_{orig}})\}$
29:                    $\delta' \leftarrow \delta' \cup \{(q_{pre_{mut}}, v[j], q_n)\}$
30:                    $\lambda' \leftarrow \lambda' \cup \{(q_{pre_{mut}}, v[j], \lambda(q_{orig}, v[j]))\}$
31:                    $q_{pre_{mut}} \leftarrow q_n$
32:                end if
33:            end for
34:            for all $(q, q_{orig}) \in newQ$ do
35:                for all $i \in I$ do          $\triangleright$ copy original behaviour for transitions from new states
36:                    $\delta' \leftarrow \delta' \cup \{(q, i, \delta(q_{orig}, i))\}$
37:                    $\lambda' \leftarrow \lambda' \cup \{(q, i, \lambda(q_{orig}, i))\}$
38:                end for
39:            end for
40:            $Mutants \leftarrow Mutants \cup \{\mathcal{M}'\}$                    $\triangleright$ add mutant
41:        end for
42:        return $Mutants$
43:    end if
44: end function
```

mutants. Let these tuples be stored in a set $IMut$. We arrange this information in an NFA created by Algorithm 6. An NFA is a 5-tuple $\langle S, s_0, \Sigma, T, F\rangle$ with states S, an initial state $s_0 \in S$, an alphabet Σ, transitions $T \subseteq S \times \Sigma \times S$, and final states $F \subseteq S$. Since it is non-deterministic, there may be several transition (s, e, s') for a source state s and a symbol e. Here, we set $\Sigma = I$, i.e. the symbols are the inputs of the original Mealy machine. Furthermore, let $S \subset Q \cup X \times IMut$ for a set of fresh symbols X. A state of the NFA is either a state of the original Mealy machine, or a new state corresponding to a mutant in

Algorithm 6 NFA-based mutant representation.

```
Input: hypothesis $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$, mutants $IMut$
Output: nondeterministic finite automaton (NFA) $NMut$
1: $NMut \leftarrow \langle S, s_0, I, T, F \rangle$ with $S \leftarrow Q, s_0 \leftarrow q_0, T \leftarrow \{(q, i, \delta(q, i)) | q \in Q, i \in I\}, F \leftarrow \{\}$
2: for all $\mathcal{M}' = \langle \mathcal{M}, (q, pI), v \rangle \in IMut$ do
3:     $s \leftarrow (x, \mathcal{M}')$ new state s.t. $s \notin S$, $S \leftarrow S \cup \{s\}$
4:     $T \leftarrow T \cup \{(q, pI, s)\}$        ▷ $s$ corresponds to state that is split
5:     for $j = 0$ to $|v| - 1$ do        ▷ add new states along sequence leading to mutation
6:         $s' \leftarrow (x, \mathcal{M}')$ new state s.t. $s' \notin S$, $S \leftarrow S \cup \{s'\}$
7:         $T \leftarrow T \cup \{(s, v[j], s')\}$
8:         if $j = |v| - 1$ then
9:             $F \leftarrow F \cup \{s'\}$        ▷ final state corresponds to mutation
10:        end if
11:        $s \leftarrow s'$
12:    end for
13: end for
```

Algorithm 7 Mutation analysis using NFA-based mutant representation.

```
Input: $test \in I^*$, $NMut = \langle S, s_0, \Sigma, T, F \rangle$
Output: Covered mutants $cMut$
1: $cMut \leftarrow \{\}$        ▷ covered mutants
2: $state \leftarrow \{s_0\}$
3: $blocked \leftarrow \{\}$
4: for $j = 0$ to $|test| - 1$ do        ▷ execute all steps of a test
5:     $next \leftarrow \{\}$
6:     for all $s \in state$ do
7:         $n \leftarrow \{\}$        ▷ partial next state for state $s$
8:         for all $t = (s, test[j], s') \in T$ s.t. $\nexists s'' : blocked(s'') = t$ do        ▷ follow unblocked $t$
9:             if $s' = (x, \mathcal{M})$ then        ▷ update $blocked$ if $t$ was added for mutant
10:                if $\exists t' : (s, t') \in blocked$ then        ▷ if current state $s$ blocks a trans.
11:                    $blocked \leftarrow (blocked \cup \{(s', blocked(s))\}) \setminus \{(s, blocked(s))\}$
12:                else        ▷ block $t$ with target state $s'$
13:                    $blocked \leftarrow blocked \cup \{(s', t)\}$
14:                end if
15:            end if
16:            $n \leftarrow n \cup \{s'\}$        ▷ add target state $s'$ to next state
17:        end for
18:        if $n = \{\} \wedge \exists t : blocked(s) = t$ then        ▷ no trans. executed and blocked a trans.
19:            $blocked \leftarrow blocked \setminus \{(s, blocked(s))\}$        ▷ unblock transition
20:        end if
21:        $next \leftarrow next \cup n$
22:    end for
23:    $cMut \leftarrow cMut \cup \{\mathcal{M}' | \exists x : (x, \mathcal{M}') \in next \wedge (x, \mathcal{M}') \in F\}$        ▷ newly covered mutants
24:    $state \leftarrow next$
25: end for
```

IMut. The generation of the NFA by Algorithm 6 works as follows. Initially, the NFA has the same structure as the original Mealy machine but without outputs. Then, we add transitions and corresponding new states for (q, pI) and v of all mutants (Line 3 to Line 11). The last state added for each of the mutants is a final state of the NFA (Line 9). Roughly speaking, a testing sequence $test \in I^*$ covers a mutant M' if it reaches the corresponding final state $(x, \mathcal{M}')$. However, we also need to account for the fact that mutants are deterministic, i.e. checking reachability in the NFA is not sufficient.

The actual mutation analysis is implemented by Algorithm 7. It implements a non-deterministic exploration of the NFA (Line 4 to Line 25), but if we enter an execution path corresponding to a mutant we temporarily block the corresponding transition (Line 9 and Line 13). Blocked transitions cannot be traversed (Line 8) and are stored in a mapping *blocked*, which maps from states to transitions. The exploration of the NFA works like the execution of several mutants in parallel. The blocking of transitions ensures that each mutant is executed at most once. Without it, we could execute multiple instances of a single mutant in parallel.

If we follow an execution path corresponding to a mutant, we update *blocked* such that the newly reached state blocks the initially blocked transition (Line 10 and Line 11). If we are in a state blocking a transition and the current input $test[j]$ did not lead to a new state, then we left the execution path of the corresponding mutant. Consequently, we unblock the transition (Line 18 and Line 19). Finally, we identify covered mutants via final states, i.e. if we visit a final state $(x, \mathcal{M}')$ with a test t, then t covers the mutant $\mathcal{M}'$ (Line 23).

5.3.2 Mutant Sampling

Since the optimisation shown above does not solve the problem completely, mutant reduction techniques need to be considered as well. Jia and Harman identify four techniques to reduce the number of mutants [20]. We use two of them: *Selective Mutation* applies only a subset of effective mutation operators. In our case, we apply only one mutation operator. With *Mutant Sampling*, only a subset of mutants is randomly selected and analysed while the rest is discarded.

Prior to sampling we collect all implicitly described mutants *IMut*. Then, we apply three types of sampling strategies in our experiments:

1. *Fraction:* this sampling strategy is parameterised by a real r. Given that, we select $\frac{|IMut|}{2^r}$ of the mutants, where each mutant is select uniformly at random from all mutants *IMut*. Hence, the resulting mutants are given by $subset(IMut, \frac{|IMut|}{2^r})$.
2. *Reduce to Minimum (redmin):* for this strategy, we partition *IMut* into sets $IMut_q$ where q is the state that is split for creating mutants in $IMut_q$. A mutant $\mathcal{M}' = \langle \mathcal{M}, (q, pI), v\rangle$ with $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda\rangle$ is in $IMut_{\delta(q,pI)}$. After partitioning, we compute $m = \min_{q\in Q}(|IMut_q|)$, the minimum number of mutants in a set. Then, we select only m mutants from these sets and combine them again, i.e. the resulting mutants are given by $\bigcup_{q\in Q} subset(IMut_q, m)$. With this strategy we achieve a uniform coverage of all states.
3. *Reduce to Mean (redmean):* this strategy is similar to the last, but we select the average number of mutants in the sets $IMut_q$, i.e. $m = \frac{\sum_{q\in Q} |IMut_q|}{|Q|}$. It may be necessary to apply this strategy if there are states q with only one access sequence, that is, $|acc(q)| = 1$. In this case, there would be no access sequence pairs for q and consequently we would not create mutants for q such that *redmin* would discard all mutants.

5.4 Further Discussion

In essence, the parameter choices, like the bound on the number of access sequences, the number of selected tests, the sample size, etc. need to take the cost of executing tests on the SUL into account. Thus, it is tradeoff between the cost of mutation analysis and testing, as a more extensive analysis can be assumed to produce better tests and thereby require fewer test executions. In the conference version of this paper, we identified two additional ways to reduce the number of mutants. We have extended the first approach and we have evaluated the second approach as well, which we previously considered as future work.

1. *Mutation analysis of executed tests:* we keep track of all tests executed on the SUL. Prior to test-case selection, these test cases are examined to determine which mutants are covered by the tests. These mutants can be discarded because we know for all executed tests t and covered mutants $\mathcal{M}_{\text{mut}}$ that $\lambda_{\text{h}}(t) = \lambda_{\text{sul}}(t)$ and $\lambda_{\text{h}}(t) \neq \lambda_{\text{mut}}(t)$ which implies $\lambda_{\text{sul}}(t) \neq \lambda_{\text{mut}}(t)$, i.e. the mutants are not implemented by the SUL. This extension prevents unnecessary coverage of already covered mutants and reduces the number of mutants to be analysed. This takes the iterative nature of learning into account as suggested in [17] in the context of equivalence testing. Previously, we checked only the tests executed for equivalence queries. We have now extended the mutation analysis to take membership queries into account as well. By doing that, we are able discard mutants whose mutations could be detected with a low number of interactions and consequently reduce computation time.
2. *Adapting to learning algorithm:* by considering the specifics of a learning algorithm, the number of access sequences could be reduced. For instance in observation-table-based learning, as in L^* [5], it would be possible to create mutants only for access sequences stored in the rows of a table. We evaluated this approach for L^* combined with counterexample processing described by Rivest and Schapire [28]. The evaluation showed that access sequences are non-uniformly distributed across states. A few states are reached by a large number of sequences while others are reached by only a few. Consequently, mutation analysis would concentrate on those states for which there are many access sequences. This turned out to be detrimental and a uniform coverage of all states should be preferred. Applying the *redmin* sampling strategy would achieve this. However, it would also discard too many mutants since it is likely that some states are reached by very few sequences in the observation table. As a result, we do not apply this approach, i.e. and we do not use data structures of learning algorithms.

6 Evaluation

In the following, we evaluate two variations of our new test-suite generation approach. We will refer to test-case generation with transition-coverage-based selection as *transition coverage*. The combination with mutation-based selection will be referred to as *mutation*. We compare these two techniques to alternatives in the literature: the partial W-method [16] and the random version of the approach discussed in [30], available at [24]. We refer to the latter as *random L & Y*. Note that this differs slightly from [14,30] in which also non-randomised tests, i.e. complete up to some bound, were generated.

In comparison to the conference version of the paper, we significantly extended the evaluation. Previously we evaluated the approach on two case studies, one implementation of a TCP server and one implementation of an MQTT broker. We now consider three implementations of TCP and four of MQTT in our evaluation. Additionally, we also evaluated the approach on nine implementations of TLS servers as well. Since we changed parts of the implementation, e.g. to optimise runtime, we redid all experiments which have already been presented. Due to optimisations, we could also perform more thorough mutation analysis in some cases and therefore changed the measurement setups slightly. The examined systems are summarised in Table 2. This table includes the number of states and inputs of the true Mealy machine model and a short description of each system. For in-depth descriptions, we refer to the publications referenced in the table.

Table 2 A short description of examined systems

System	#States	#Inputs	Short description
TCP servers			
Ubuntu	57	12	Models of TCP server/client implementations from three different vendors have been learned and analysed by Fiterău-Broştean et al. [14]. We simulated the server models available at [34]
Windows	38	13	
BSD	55	13	
MQTT brokers			
emqttd 1.0.2	18	9	Models of an MQTT [6] broker interacting with two clients. We discussed the learning setup in [33]
VerneMQ 0.12.5p4	17	9	
HBMQTT 0.7.1	17	9	
Mosquitto 1.4.9	18	9	
TLS servers			
GnuTLS 3.3.8	16	11	Models of TLS servers learned by de Ruiter and Poll [11], which are available at [12]
GnuTLS 3.3.12	9	12	
miTLS 0.1.3	6	8	
NSS 3.17.4	8	8	
OpenSSL 1.0.1j	11	7	
OpenSSL 1.0.1l	10	7	
OpenSSL 1.0.2	7	7	
RSA BSAFE for C 4.0.4	9	8	
RSA BSAFE for Java 6.1.1	6	8	

6.1 Measurement Setup

To objectively evaluate randomised conformance testing, we investigate the probability of learning the correct model with a limited number of interactions with the SUL, i.e. only a limited number of tests may be executed. We generally base the cost of learning on the number of executed inputs rather than on the number of tests/resets. This decision follows from the observation that resets in the target application area, communication protocols, can be done fast (simply start a new session), whereas timeouts and quiescent behaviour cause long test durations [11,33]. Note that we take previously learned models as SULs. These models are given in a textual format specifying Mealy machines comprising the available inputs, outputs, transitions, and states. To simulate them, we built a test driver which produces outputs corresponding to given input sequences. Consequently, we still take a black-box view in which we interact with the simulated systems only via testing. As compared to testing of the actual systems, simulation of models allows fast test execution enabling a thorough evaluation.

We refer to one execution of a learning algorithm as a learning run, i.e. a learning run consists of several rounds which are concluded by an equivalence query that is carried out via testing. To estimate the probability of learning the correct models with a given setup, we perform 50 learning runs and calculate the relative frequency of learning the correct model.

In the following, we refer to such an execution of 50 learning runs as a single experiment. We deem learning reliable if the corresponding probability estimation is equal to one. Note that given the expected number of states of each system, we can efficiently determine whether the correct model has been learned, since the L^* algorithm guarantees that learned models are minimal with respect to the number of states [5].

In order to find a lower limit on the number of tests required by each method to work reliably, we bounded the number of tests executed for each equivalence query and gradually increased this bound. Once all learning runs of an experiment succeeded we stopped this procedure. As in Sect. 4, we refer to this bound also as n_{sel} because it defines the number of tests selected for execution. For learning with the partial W-method [16] we gradually increased the depth parameter implemented in LearnLib [19] until we learned the correct model. Since this method does not use randomisation, we did not run repeated experiments and report the measurement results for the lowest possible depth-parameter value.

As all algorithms can be run on standard notebooks, we will only exemplarily comment on runtime. For a fair comparison, we evaluated all equivalence-testing approaches in combination with the same learning algorithm, i.e. L^* with Rivest and Schapire's counterexample-handling implemented by LearnLib 0.12 [19].

6.2 TCP

The number of tests and steps required to reliably learn the different TCP-servers are given in Table 3. In order to perform these experiments, we generated 200,000 tests as a basis for the test selection of the approaches *mutation* and *transition coverage*. From these tests, we selected the number of tests given in the third row of Table 3 to perform each equivalence query. For *random L & Y* we generated the number of tests given in the third row but we did not perform coverage-based selection from the tests generated by this approach. For the partial W-method this row includes the depth-parameter value which determines the set of tests to be executed for each equivalence query.

The test-case generation with Algorithm 2 has been performed with parameters $maxSteps = 60$, $p_{\text{retry}} = 0.95$, $p_{\text{stop}} = 0.05$, and $l_{\text{infix}} = 6$. The chosen parameters for *mutation* selection are $k = 2$ (length of distinguishing sequence) and $n_{\text{acc}} = 100$. Due to our implementation of *acc* (see Sect. 5) which may return only a fraction of access sequences, setting n_{acc} to a value as large as 100 has the effect that no access sequences are discarded. Since we sample mutants afterwards, we set it to such a large value. We performed sampling by first applying the *redmin* sampling strategy and then *fraction* sampling with $r = 1$. Note that we used the same setup for all TCP implementations. We only varied the bound n_{sel} on the number of equivalence tests. With that, we want to demonstrate that it is possible to learn several systems from the same application domain with similar setup. In practice, it is therefore possible to learn a model with a conservatively chosen setup, i.e. with a large number of tests, and then use this model to fine-tune parameter settings. These parameters may then be used to learn further models with similar structure.

As noted above, we executed 50 learning runs for each experiment. During one run, the number of tests for a single equivalence query (there may be several in one run), is bounded by the number given in Row 3 of Table 3. We collected data on the overall number of tests executed for equivalence/membership queries as well as the overall number of inputs executed during those tests. The table shows these values averaged over all 50 runs of an experiment, where *eq* stands for equivalence test queries and *mem* stands for membership queries. We also show further statistics with respect to the number of test steps executed for equivalence

Table 3 Performance measurements for learning TCP-server models

	Mutation	Transition coverage	Partial W-method	Random L & Y
Ubuntu				
n_{sel}/depth	3500	5000	2	44,000
Mean # tests [eq.]	4074	5628	793,939	65,716
Mean # steps [eq.]	154,551	350,678	7,958,026	703,315
Median # steps [eq.]	151,758	334,147		643,785
Q1 # steps [eq.]	141,497	322,584		588,320
Q3 # steps [eq.]	164,089	346,430		761,456
Min. # steps [eq.]	132,953	313,800		519,250
Max. # steps [eq.]	226,119	612,634		1,338,526
Mean # tests [mem.]	9015	9237	13,962	11,896
Mean # steps [mem.]	120,195	128,276	147,166	138,340
BSD				
n_{sel}/depth	1500	2000	5	58,000
Mean # tests [eq.]	2037	2361	2,105,585,114	96,224
Mean # steps [eq.]	91,089	152,301	30,435,822,650	1,069,553
Median# steps [eq.]	86,031	147,513		1,002,069
Q1 # steps [eq.]	77,142	139,952		857,912
Q3 # steps [eq.]	99,824	156,330		1,225,874
Min. # steps [eq.]	69,771	131,788		728,148
Max. # steps [eq.]	152,133	270,108		1,985,223
Mean # tests [mem.]	8989	9437	15,608	12,219
Mean # steps [mem.]	132,126	141,588	170,416	146,893
Windows				
n_{sel}/depth	3500	2500	2	40,000
Mean # tests [eq.]	4337	3006	521,635	49,594
Mean # steps [eq.]	144,074	178,985	4,597,896	514,458
Median# steps [eq.]	131,750	167,658		492,830
Q1 # steps [eq.]	125,873	158,452		455,614
Q3 # steps [eq.]	158,914	195,210		555,076
Min. # steps [eq.]	120,173	152,442		419,910
Max. # steps [eq.]	210,253	298,169		839,380
Mean # tests [mem.]	5939	5999	7919	6865
Mean # steps [mem.]	65,137	63,547	67,834	68,563

testing as we regard this to be the most important measure of performance. In addition to the complete information given in the table, we summarise the most important measures of performance in bar charts. Figure 6 provides an overview of the average number of required equivalence test steps and Fig. 7, provides an overview of the required n_{sel}, i.e. the minimum number equivalence tests to learn reliably.

In Table 3, we see that the average number of tests and steps required for membership queries is roughly the same for all techniques. This is what we expected as the same learning algorithm is used in all cases, but the numbers shall demonstrate that techniques requiring

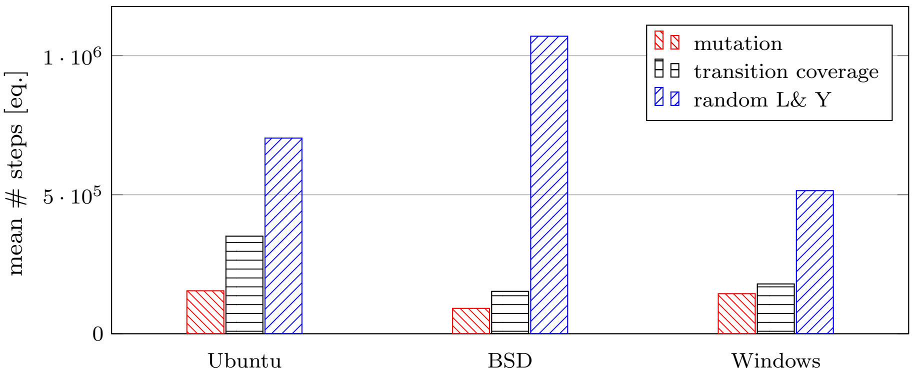


Fig. 6 Average number of equivalence test steps required to reliably learn TCP models

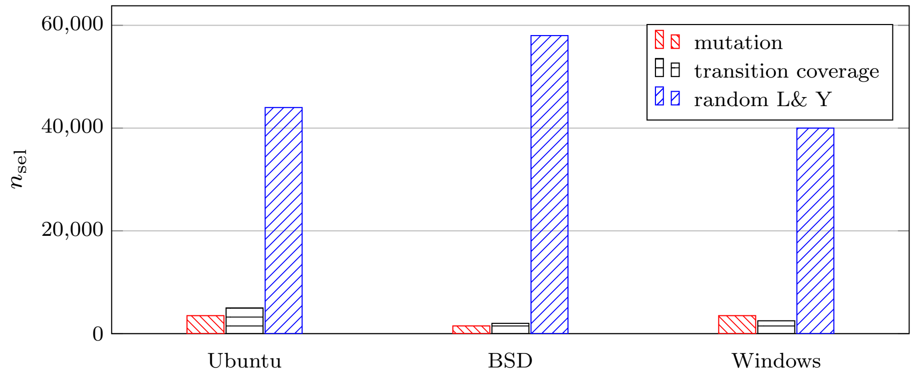


Fig. 7 Minimum number of tests per equivalence query to reliably learn TCP models

less equivalence tests do not trade membership for equivalence tests. It can be considered a coincidence that learning with the partial W-method requires more membership queries while also requiring more equivalence tests. With this out of the way, we can concentrate on equivalence testing.

Figure 6 shows that *mutation* pays off in this experiment as it performs best for all different systems. It performs for instance significantly better than *transition coverage* for Ubuntu, which requires 2.27 times as many equivalence test steps on average. The average cost of test selection is approximately 324 s for *mutation* and 9 s for *transition coverage*. However, considering the large savings in actual test execution, *mutation* performs better. Compared to the conference version of this paper, we generate more tests and perform less aggressive mutant sampling, therefore we see an increase in runtime. The small performance gain of *mutation* over *transition coverage* for Windows shows that our mutation analysis may not add value in some situations.

Considering the minimum required number of equivalence tests n_{sel} depicted in Fig. 7, *mutation* and *transition coverage* show similar performance. Given that *transition coverage* executed more test steps suggests that it favours longer tests than *mutation*. Since both techniques require relatively few tests, they are applicable in setups, where reset operations are computationally expensive.

We also evaluated *random L & Y* with a middle sequence of expected length 3 for BSD and Ubuntu and length 4 for Windows (similar to [14] where 4 was used). For this setup, *random L & Y* requires significantly more steps and tests than both alternatives (see Figs. 6 and 7). There may be more suitable parameters, however, which would improve the performance of *random L & Y*. Moreover, the implementation also offers test generation for complete conformance testing up to some bound which may be beneficial as well. Nevertheless, the model structure of the TCP-servers seems to be well-suited for our approach.

All randomised approaches outperformed the partial W-method. For instance for Ubuntu, *mutation* reduces the number of equivalence test steps by a factor of 51.5 on average (Row 5 of Table 3). Taking the membership queries into account (Row 11 of Table 3), the overall cost of learning is reduced by a factor of 29.5. Looking at the average number of executed equivalence tests (Row 4 of Table 3) rather than at the test steps, we see an even larger reduction. The relative gap between equivalence tests shows a reduction by a factor of about 195 on average. This is an advantage of our approach as we can flexibly control test length and thereby account for systems with expensive resets. The enormously large number of tests required for the partial W-method for BSD highlights a problem of complete conformance testing. Increasing the depth parameter causes an exponential growth of the number of tests. In practice executing such a large number of tests, as required to correctly learn the BSD model, would be infeasible. Pure random testing is not a viable choice in this case as well. A learning experiment with Ubuntu showed that it is infeasible to reliably learn correct models. We executed 1,000,000 tests with a uniformly distributed length between 1 and 50 and succeeded in learning correctly in only 4 out of 50 runs.

Finally, we want to investigate the distribution of equivalence test steps. The gap between the first quartile $Q1$ and the third quartile $Q3$ is generally relatively small. Therefore, half of the runs of an experiment require roughly the same number of test steps. In other words, we see a uniform performance across runs. In general, the ratio between minimum and maximum number of steps is not very large as well. It is largest for *random L & Y* and BSD with a value of 2.73. This property of uniform performance of the randomised approaches can be explained by the following observations. Only few tests are usually required to find a counterexample in the early phases of learning while the last equivalence queries require thorough testing. Put differently, a large portion of the state space can be explored with a small number of tests. Since there are no extreme outliers, we give only average numbers of equivalence test steps for the next case studies.

6.3 MQTT

The number of tests and steps required to reliably learn models of four different MQTT brokers are given in Table 4. In order to perform these experiments, we used largely the same setup as for the TCP experiments, but generated only 50,000 tests as a basis for selection. Additionally, we decreased the maximum test length *maxSteps* to 40 and changed the parameter r of the *fraction* sampling strategy to 0, i.e. we effectively only applied the *redmin* strategy. In contrast to the conference version of the paper we set $k = 2$ instead of $k = 3$, i.e. the lengths of the distinguishing sequences are shorter. This allows less aggressive sampling and serves to show that the same parameters as for TCP may be used while still achieving satisfactory performance. We applied *random L & Y* with a middle sequence of expected length 3. We also provide a graphical overview of the most important performance measure, the average number of equivalence test steps, in Fig. 8.

Table 4 Performance measurements for learning MQTT-broker models

	Mutation	Transition coverage	Partial W-method	Random L & Y
emqttd				
n_{sel}/depth	175	200	2	1085
Mean # tests [eq.]	253	253	72,308	1664
Mean # steps [eq.]	11,058	11,822	487,013	11,857
Mean # tests [mem.]	1647	1603	1808	1683
Mean # steps [mem.]	13,655	12,942	11,981	12,005
HBMQTT				
n_{sel}/depth	200	325	2	4350
Mean # tests [eq.]	255	374	49,860	5173
Mean # steps [eq.]	10,986	17,212	334,298	35,781
Mean # tests [mem.]	1067	1044	1033	1111
Mean # steps [mem.]	9116	8257	6133	7513
Mosquitto				
n_{sel}/depth	150	100	2	650
Mean # tests [eq.]	187	132	59,939	1036
Mean # steps [eq.]	8201	6102	400,781	7240
Mean # tests [mem.]	1309	1372	1355	1378
Mean # steps [mem.]	10,686	10,218	8623	9271
VerneMQ				
n_{sel}/depth	125	125	2	1200
Mean # tests [eq.]	183	177	56,609	1692
Mean # steps [eq.]	7983	8245	375,263	11,947
Mean # tests [mem.]	1295	1336	1279	1350
Mean # steps [mem.]	10,410	10,063	7985	9193

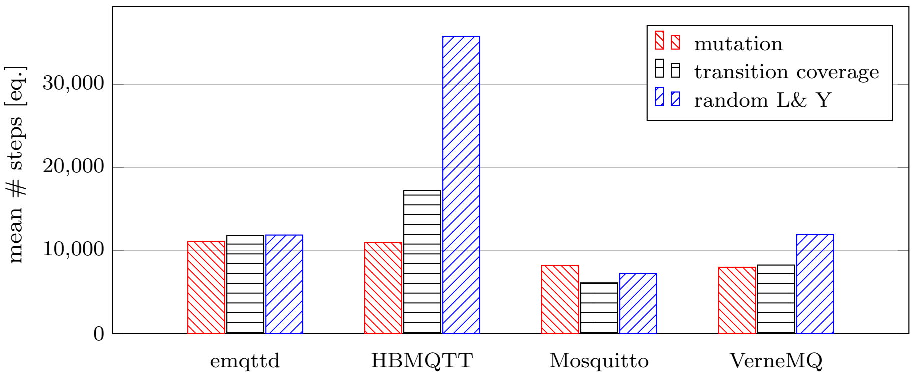


Fig. 8 Average number of equivalence test steps required to reliably learn MQTT models

As before, Table 4 shows that the randomised approaches outperform the partial W-method. Considering test execution time in non-simulated setups highlights that applying such a randomised approach may save significant amounts of time. In the original test setup [33],

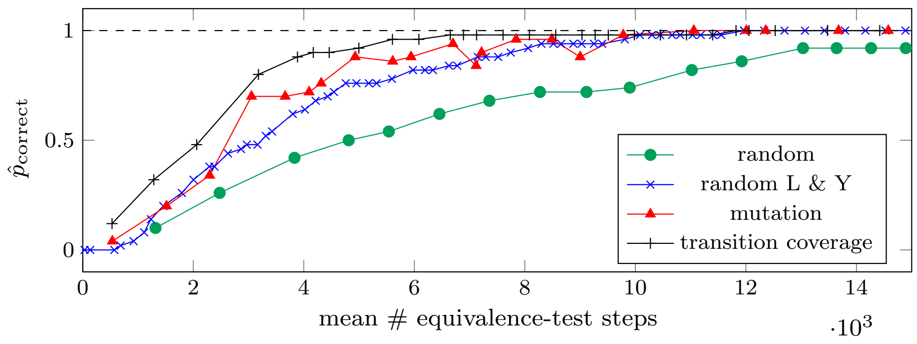


Fig. 9 Average number of equivalence-test steps required to reliably learn the correct emqttd-broker model

execution time was, e.g., heavily affected by network communication. The execution of a single test step while learning VerneMQ models took 600 ms (waiting for outputs from two clients with a timeout of 300 ms per client). As a result, it would take approximately $(375{,}263 + 7985) \cdot 0.6\,\text{s} \approx 63.9\,\text{h}$ to learn a model of VerneMQ with the partial W-method. Applying the mutation-based approach *mutation* on the other hand would require $(7983 + 10{,}410) \cdot 0.6\,\text{s} \approx 3.1\,\text{h}$ while still learning reliably. Since these calculations take membership and equivalence queries into account, this gives us a reduction of learning time by a factor of about 20.6. We see varying but large reductions for all other MQTT brokers as well.

Figure 8 shows that all three of the randomised approaches perform similarly well with no clear winner. For instance for emqttd, *random L & Y* performs best in terms of executed test steps, if we take membership queries into account (Row 7 of Table 4). The *transition coverage* approach requires the least number of steps for the Mosquitto model, but performs only slightly better than *random L & Y*. However, *mutation* also performs well for this example. We assume that our mutation-based approach produces good results in general, but will not be the best-performing solution in all circumstances.

Figure 9 shows how reliably each of the three approaches learned the emqttd model. Additionally, it also provides a comparison to undirected random testing in which each input is chosen uniformly at random from all available inputs and where the test length is uniformly distributed between 1 and 50. As noted at the beginning of this section, we say that a model is learned reliably if it is learned correctly with a high probability. We estimate the probability of learning correctly by computing the relative number of successful learning runs, i.e. runs in which the correct model was learned. This estimation is denoted by $\hat{p}_{\text{correct}}$ as given by the y-axis in Fig. 9. The x-axis gives the required number of equivalence test steps. We see in the figure that the increase of both *random L& Y* and *transition coverage* is more stable than that of *mutation*. Although allowing for more equivalence test steps generally increases reliability of learning, we also see slight drops for *mutation*. The reason for this behaviour is that we are able to cover only a small portion of the mutants with a low number of tests such that we may select tests covering irrelevant mutants. Covering all transitions on the other hand is simpler and *random L & Y* follows a rather different approach. However, if we allow for sufficiently many tests to be selected and executed, these drops can be expected to disappear. Despite these drops, *mutation* requires slightly less steps to learn reliably. Furthermore, we see that random testing may successfully be applied but it is less reliable at producing correct results.

6.4 TLS

For the last case study, we learn models of nine TLS-server implementations [11]. The models were originally learned by de Ruiter and Poll while taking domain-specific knowledge into account. They stopped executing a test once the connection between TLS server and test harness had been closed. This is reflected in the structure of the models such that we may use shorter tests. Therefore, we set the following parameters for test generation: $maxSteps = 20$, $p_{\text{retry}} = 0.95$, $p_{\text{stop}} = 0.05$, and $l_{\text{infix}} = 3$, i.e. we reduced the bound on test length. Since there are nine implementations with varying structure (see number of states in Table 2), we decided to perform a thorough coverage analysis by generating 300,000 tests as a basis for test selection. We used the same mutation parameters as before but a different sampling strategy. Some of the states of the considered models were reached by only one access sequence such that no mutants would be produced for these states. As a result, *redmin* would discard all mutants produced for the other states as well. Instead, we applied the *redmean* strategy and *fraction* sampling with $r = 1$.

Since we have seen that membership queries are not affected by equivalence testing, we give only measurement results for equivalence testing. These are listed in Table 5 and like for all other models, we provide an overview of the average number of executed equivalence test steps shown in Fig. 10. The first issue to notice is that for *transition coverage* and the OpenSSL models, we would need to set the equivalence test bound n_{sel} to a value larger than 10,000. For this reason, we stopped at this point and deemed learning infeasible with this setup. The same holds for *random L& Y* and GnuTLS 3.3.8 where we performed experiments with up to 20,000 tests because the generated tests were shorter than the tests generated with Algorithm 2. The *mutation* approach on the other hand was successful for all models.

In Table 5, we notice favourable performance of the partial W-method. It even requires the lowest number of test steps for OpenSSL 1.0.2. This stands in stark contrast to previous observations but is simply caused by the low depth parameter required for TLS models. It actually highlights the effect of the depth parameter. To correctly learn the GnuTLS 3.3.8 model, a depth parameter of 3 is required. For this model, *mutation* needs three orders of magnitude less test steps. If we were to choose 2 as parameter value for all other models to be on the "safe side", the performance of the partial W-method would drastically drop. Note that we usually do not know the required depth in practice.

Comparing the randomised approaches in Fig. 10, we see that *mutation* generally performs well. It is not the best performing solution in all cases, though. We made this observation above for MQTT and here we see it again. Hence, *mutation* may be better suited to changing environments. In comparison to the other approaches, it shows the worst performance for miTLS, for which it requires three times as many test steps as *transition coverage*. However, it performs best for other models, like for OpenSSL 1.0.1j. Note that it does not show extreme outliers as well. The different versions of OpenSSL for example cannot be efficiently learned with *transition coverage*.

It should be noted that the non-randomised version of *random L & Y* discussed in [30] can be assumed to perform best for the models requiring a depth of 1. The reason for that is that it implements an effective method for computing distinguishing sequences. Moreover, the partial W-method already performs well for these models.

Table 5 Performance measurements for learning TLS-server models

	Mutation	Transition coverage	Partial W-method	Random L & Y
GnuTLS 3.3.8				
n_{sel}/depth	100	200	3	> 20,000
Mean # tests [eq.]	182	283	709,845	–
Mean # steps [eq.]	2888	3513	4,979,346	–
GnuTLS 3.3.12				
n_{sel}/depth	100	300	1	7600
Mean # tests [eq.]	141	333	1354	8776
Mean # steps [eq.]	2441	5819	5815	52,536
miTLS 0.1.3				
n_{sel}/depth	300	100	1	700
Mean # tests [eq.]	347	122	1017	884
Mean # steps [eq.]	5913	2027	4508	4939
NSS 3.17.4				
n_{sel}/depth	200	100	1	1000
Mean # tests [eq.]	274	138	1428	1357
Mean # steps [eq.]	4206	2115	5970	7932
OpenSSL 1.0.1j				
n_{sel}/depth	1100	> 10,000	2	12,400
Mean # tests [eq.]	1209	–	13,853	15,392
Mean # steps [eq.]	16,635	–	78,615	100,332
OpenSSL 1.0.1l				
n_{sel}/depth	800	> 10,000	2	15,300
Mean # tests [eq.]	864	–	12,666	18,332
Mean # steps [eq.]	11,566	–	68,524	115,122
OpenSSL 1.0.2				
n_{sel}/depth	500	> 10,000	1	500
Mean # tests [eq.]	529	–	916	690
Mean # steps [eq.]	7934	–	3621	3970
RSA BSAFE for C 4.0.4				
n_{sel}/depth	100	5000	1	1100
Mean # tests [eq.]	125	5831	980	1369
Mean # steps [eq.]	1656	106,563	4188	8209
RSA BSAFE for Java 6.1.1				
n_{sel}/depth	200	100	1	600
Mean # tests [eq.]	254	137	960	873
Mean # steps [eq.]	4235	2247	3877	4793

6.5 Discussion, Limitations and Threats to Validity

The measurements shown and discussed above suggest that our mutation-based test-selection pays off. *Mutation* performed very well in the TCP case study, but did not perform best in all other cases. Still, it generally showed good performance in comparison to the other

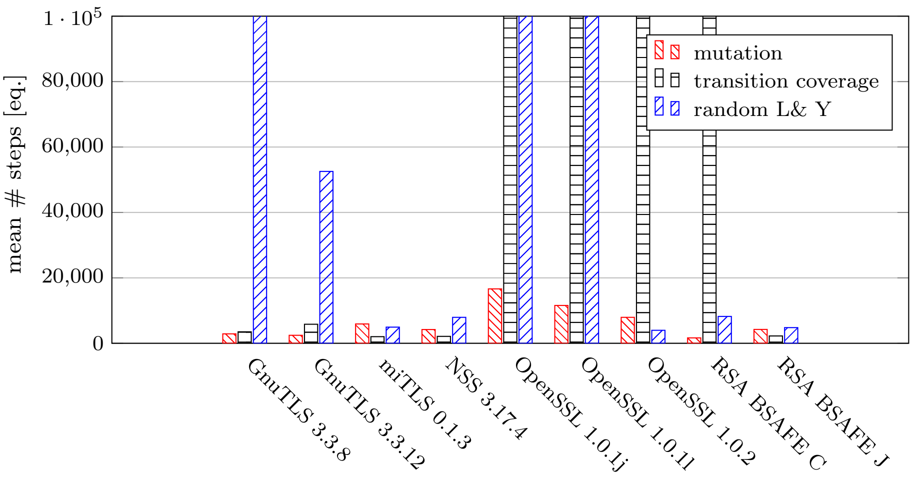


Fig. 10 Average number of equivalence test steps required to reliably learn TLS models

randomised approaches. There are, e.g., systems for which *transition coverage* performs slightly better, i.e. mutation-based selection may not add value for these systems. It should be noted that a crucial aspect of *mutation* is that it uses Algorithm 2 for test generation. Initially, we generated test cases purely randomly. These tests, however, would fail to reach most of the mutations. The coverage-directed generation of Algorithm 2 mitigates this issue. Although we extended the evaluation in comparison to the conference version of this paper, it is still limited to variations of three types of systems. Our approach may fail for other systems, especially from other application domains. However, we generally target communication protocols as application domain.

We mainly compared *mutation* with the partial W-method. The total cost of learning in our setting is given by the average combined number of test steps for equivalence and membership queries. *Mutation* improves upon the partial W-method with respect to this measure by at least one order of magnitude for all MQTT and TCP models. It may be argued that the comparison is unfair because the partial W-method provides guarantees while *mutation* aims at optimising coverage. In the considered setup, however, *mutation* provides guarantees as well. Since the automata learned with L^* are minimal [5], we can state: if we learn a model, then it is either correct or the SUL has more states than the learned model. The (partial) W-method performed with depth d provides the stronger guarantee: if we learn a model with k states, then it is either correct or the SUL has more than $k + d$ states [16]. Given the difference in guarantees, we bounded the number of equivalence test steps for the partial W-method in one experiment and tried learning the Ubuntu TCP-server model. Testing with the partial W-method with depth 2 while bounding the number of steps by 226,119 (the maximum number of steps required by *mutation*), resulted in a model with 27 states. Hence, the guarantee given would be that the model is either correct or the system has more than 29 states. Strictly speaking, the guarantee would actually be weaker because we did not execute all tests prescribed by the technique. If we learn with *mutation*, we come up with a lower bound on the number of states of 57, as we learn the true model. Thus, conditioned on allowed test execution time, learning with *mutation* is able to give stronger guarantees in learning.

An advantage of the randomised approaches is that they can be controlled more flexibly through parameters. However, a large number of parameters also increases the difficulty of finding suitable parameters. This holds for all performed experiments. There may be better parameters for *mutation* as well as for *random L & Y*. In general, we tried to find parameters that would work for one of the MQTT/TCP/TLS models and use these settings for all the other models as well. Since we developed the *mutation* technique, there may be a bias affecting the learning performance. Put differently, it is likely that we are able to find well-suited parameters for our approach but not for *random L & Y*. For this reason, performance gains of *mutation* over *random L & Y* may be smaller in general.

Another risk related to the large parameter space of *mutation* is that it may be possible that only a small portion of the parameters actually produces good results. To limit the risk that we found good parameters by chance, we used the same parameter settings for all systems of a certain type. The TLS case study shows that the same parameter settings may work nicely for various different systems. Still, finding parameters for *mutation* is likely to be the most difficult.

Another benefit of randomised approaches is that randomness introduces variability which may help. For the BSD case study, we needed to set the depth of the partial W-method to 5. In principle, this implies for *mutation* that mutants should be created with distinguishing sequences of length $k = 5$. But we set $k = 2$ and still learned successfully. This may be explained by the fact that if we cover a large number of mutants with $k = 2$, we will with high probability cover mutants with $k > 2$ as well. Additionally, variability in tests may help to explore the search space more thoroughly. Pure random testing did not perform well, though. Without any form of directed testing, we fail to reach relevant portions of the search space.

A shortcoming of *mutation* is that mutation analysis is computationally intensive although we spend effort optimising it. Therefore, we target moderately-sized systems with this approach. As *random L & Y* generates tests much more efficiently, it can be applied for larger systems as well. It has, e.g., been used to learn a system with more than 3000 states [30]. Applying *mutation* to systems with significantly more than 100 states would likely not pay off. We would have to perform aggressive mutant sampling which would negatively impact the effectiveness of mutation-based selection.

In general, we did not evaluate the influence of mutant sampling. We have observed during development that sampling is detrimental to performance if we discard too many mutants. However, a thorough evaluation would be necessary to explore the effects and limitations of sampling.

7 Conclusion

We presented a fast test-case generation technique which accompanied with appropriate test-case selection yields effective test suites. In particular, we further motivated and described a fault-based test selection approach with a fault model tailored towards learning. We performed various experiments in the domain of communication protocols. They showed that it is possible to reliably learn system models with a significantly lower number of test cases, as compared to complete conformance testing with, e.g., the partial W-method [16].

A potential drawback of our approach, especially of split-state-based test selection, is the large number of parameters, which according to our experience heavily influence learning performance. However, our evaluation showed that it is possible to efficiently learn several similar models with the same parameters. Additionally, mutation-based selection applies mutant sampling, thus it is of interest to determine the influence of sampling and whether corresponding observations made for program mutation [20] also hold for FSM mutation.

We conclude that mutation-based test-suite generation is a promising technique for conformance testing in active automata learning. Despite initial success, we believe that it could show its full potential for testing more expressive types of models like extended finite state machines [8]. This would enable the application of more comprehensive fault models. Finally, alternatives to the simple greedy test-selection may also provide benefits.

Acknowledgements Open access funding provided by Graz University of Technology. This work was supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". We would also like to thank the developers of LearnLib and of the test-case generator available at [24]. Furthermore, we would like to thank researchers making learned models available for others, especially the contributors of http://automata.cs.ru.nl, a collection of automata benchmarks. During the development of the presented approach, all models used in Sect. 6 have been integrated into this collection. The schedule says that it was received on 8 October 2018, but we initially submitted the first version on 29 November 2017.

References

1. Aichernig, B.K., Auer, J., Jöbstl, E., Korosec, R., Krenn, W., Schlick, R., Schmidt, B.V.: Model-based mutation testing of an industrial measurement device. In: Seidl, M., Tillmann, N. (eds.) Tests and Proofs-8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24–25, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8570, pp. 1–19. Springer (2014). https://doi.org/10.1007/978-3-319-09099-3_1
2. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W., Schlick, R., Tiran, S.: Killing strategies for model-based mutation testing. J. Softw. Test. Verif. Reliab. (STVR) **25**(8), 716–748 (2015). https://doi.org/10.1002/stvr.1522
3. Aichernig, B.K., Mostowski, W., Mousavi, M.R., Tappler, M., Taromirad, M.: Model learning and model-based testing. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits-International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24–27, 2016, Revised Papers, Lecture Notes in Computer Science, vol. 11026, pp. 74–100. Springer (2016). https://doi.org/10.1007/978-3-319-96562-8_3
4. Aichernig, B.K., Tappler, M.: Learning from faults: mutation testing in active automata learning. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NASA Formal Methods—9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16–18, 2017, Proceedings, LNCS, vol. 10227, pp. 19–34 (2017). https://doi.org/10.1007/978-3-319-57288-8_2
5. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
6. Banks, A., Gupta, R.: MQTT Version 3.1.1. OASIS Standard. Latest version. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html (2014)
7. 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, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 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
8. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. Form. Asp. Comput. **28**(2), 233–263 (2016). https://doi.org/10.1007/s00165-016-0355-5
9. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **4**(3), 178–187 (1978). https://doi.org/10.1109/TSE.1978.231496
10. Combe, D., de la Higuera, C., Janodet, J.: Zulu: an interactive learning competition. In: Yli-Jyrä, A., Kornai, A., Sakarovitch, J., Watson, B.W. (eds.) Finite-State Methods and Natural Language Processing, 8th International Workshop, FSMNLP 2009, Pretoria, South Africa, July 21–24, 2009, Revised Selected Papers, Lecture Notes in Computer Science, vol. 6062, pp. 139–146. Springer (2009). https://doi.org/10.1007/978-3-642-14684-8_15

11. 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. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter (2015)
12. de Ruiter, J., Poll, E.: TLS—learned models. http://www.cs.ru.nl/J.deRuiter/download/usenix15.zip. Accessed 15 Nov 2017 (2015)
13. Fabbri, S.C.P.F., Delamaro, M.E., Maldonado, J.C., Masiero, P.C.: Mutation analysis testing for finite state machines. In: 5th International Symposium on Software Reliability Engineering, ISSRE 1994, Monterey, CA, USA, November 6–9, 1994, pp. 220–229. IEEE (1994). https://doi.org/10.1109/ISSRE.1994.341378
14. Fiterău-Broştean, P., Janssen, R., Vaandrager, F.W.: Combining model learning and model checking to analyze TCP implementations. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification—28th International Conference, CAV 2016, Toronto, ON, Canada, July 17–23, 2016, Proceedings, Part II, Lecture Notes in Computer Science, vol. 9780, pp. 454–471. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_25
15. Fiterău-Broştean, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F.W., Verleg, P.: Model learning and model checking of SSH implementations. In: Erdogmus, H., Havelund, K. (eds.) Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10–14, 2017, pp. 142–151. ACM (2017). https://doi.org/10.1145/3092282.3092289
16. Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., Ghedamsi, A.: Test selection based on finite state models. IEEE Trans. Softw. Eng. **17**(6), 591–603 (1991). https://doi.org/10.1109/32.87284
17. Howar, F., Steffen, B., Merten, M.: From ZULU to RERS—lessons learned in the ZULU challenge. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification, and Validation—4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18–21, 2010, Proceedings, Part I, Lecture Notes in Computer Science, vol. 6415, pp. 687–704. Springer (2010). https://doi.org/10.1007/978-3-642-16558-0_55
18. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification—5th International Conference, RV 2014, Toronto, ON, Canada, September 22–25, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8734, pp. 307–322. Springer (2014). https://doi.org/10.1007/978-3-319-11164-3_26
19. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib—a framework for active automata learning. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification—27th International Conference, 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
20. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011). https://doi.org/10.1109/TSE.2010.62
21. Lee, D., Yannakakis, M.: Testing finite-state machines: State identification and verification. IEEE Trans. Comput. **43**(3), 306–320 (1994). https://doi.org/10.1109/12.272431
22. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines—a survey. Proc. IEEE **84**(8), 1090–1123 (1996). https://doi.org/10.1109/5.533956
23. 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
24. Moerman, J.: Yannakakis—test-case generator. https://gitlab.science.ru.nl/moerman/Yannakakis. Accessed 30 Nov 2016 (2015)
25. Niese, O.: An integrated approach to testing complex systems. Ph.D. thesis, Dortmund University of Technology (2003)
26. Peled, D.A., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Wu, J., Chanson, S.T., Gao, Q. (eds.) Formal Methods for Protocol Engineering and Distributed Systems, FORTE XII / PSTV XIX'99, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), October 5–8, 1999, Beijing, China, IFIP Conference Proceedings, vol. 156, pp. 225–240. Kluwer (1999)
27. Pretschner, A.: Defect-based testing. In: Dependable Software Systems Engineering, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 40, pp. 224–245. IOS Press (2015). https://doi.org/10.3233/978-1-61499-495-4-224
28. 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

29. Shahbaz, M., Groz, R.: Inferring Mealy machines. In: Cavalcanti, A., Dams, D. (eds.) FM 2009: Formal Methods, Second World Congress, 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
30. Smeenk, W., Moerman, J., Vaandrager, F.W., Jansen, D.N.: Applying automata learning to embedded control software. In: Butler, M.J., Conchon, S., Zaïdi, F. (eds.) Formal Methods and Software Engineering—17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3–5, 2015, Proceedings, Lecture Notes in Computer Science, vol. 9407, pp. 67–83. Springer (2015). https://doi.org/10.1007/978-3-319-25423-4_5
31. Steffen, B., Howar, F., Merten, M.: Introduction to active automata learning from a practical perspective. In: Bernardo, M., Issarny, V. (eds.) Formal Methods for Eternal Networked Software Systems—11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13–18, 2011. Advanced Lectures, Lecture Notes in Computer Science, vol. 6659, pp. 256–296. Springer (2011). https://doi.org/10.1007/978-3-642-21455-4_8
32. Tappler, M.: mut-learn—randomised mutation-based equivalence testing. https://github.com/mtappler/mut-learn. Accessed 07 Dec 2016 (2017)
33. 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
34. TCP models. https://gitlab.science.ru.nl/pfiteraubrostean/tcp-learner/tree/cav-aec/models. Accessed: 14 Nov 2016 (2016)
35. Vasilevskii, M.P.: Failure diagnosis of automata. Cybernetics **9**(4), 653–665 (1973). https://doi.org/10.1007/BF01068590