

Learning from Faults: Mutation Testing in Active Automata Learning

Bernhard K. Aichernig and Martin Tappler^(✉)

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

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 one experiment by a factor of more than twenty.

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

1 Introduction

Since Peled et al. [21] 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. Active learning of automata in the minimally adequate teacher (MAT) framework, as introduced by Angluin [2], assumes the existence of a teacher. In the non-stochastic setting, 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.

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. [21], 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 [30] and Chow [6].

This method is also referred to as W-method and there exist optimisations of it, like the partial W-method [11] or an approach by Lee and Yannakakis [16], but all have the same worst-case complexity [4]. 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” [4].

In practice, there is limited time for testing and thereby also for learning. The ZULU challenge [7] addressed this issue by limiting the number of tests to be executed [12]. 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” [12].

In this paper we propose an implementation of equivalence queries based on mutation testing [15], more specifically on model-based mutation testing [1]. 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 [14], we will mainly compare it to the partial W-method [11] 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. [26].

We target systems which can be modelled with a moderately large number of states, i.e. with up to fifty states. This restriction is necessary, because mutation analysis is generally 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. The rest of this paper is structured as follows. Section 2 discusses related work and Sect. 3 introduces preliminaries. The main parts, the test-suite generation approach and its evaluation, are presented in Sects. 4 and 5. We conclude the paper in Sect. 6. The implementation used in our evaluation is available at [27].

2 Related Work

We address conformance testing in active automata learning. Hence, there is a relationship to the W-method [6, 30] and the partial W-method [11], two conformance testing methods implemented in LearnLib [14]. 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. 4.3.

We combine model-based mutation testing and random testing, which we discussed in previous work [1]. 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. noted that it is necessary to find counterexamples with few tests for automata learning to be practically applicable [12]. We generally follow this approach. Furthermore, one of the heuristics described in [12] is based on Rivest and Schapire’s counterexample processing [23], similar to the fault model discussed in Sect. 4.3. More recent work in this area has been performed by Smeenk et al. [26], 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 [17]. 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 [10].

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 [10, 18, 24, 28]. In addition to that, the Java-library LearnLib [14] 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 is a finite set 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 surjective. 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.

Notational Conventions. 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.

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 $\lambda(s) = \lambda(q_0, s)$. For state q the set $acc(q) = \{s \in I^* \mid \delta(q_0, s) = q\}$ 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 [13].

Finally we need a basis for determining whether two Mealy machines are equivalent. Equivalence is usually defined with respect to outputs [10], i.e. two deterministic Mealy machines are equivalent if they produce the same outputs for all input sequences. A Mealy machine $\langle Q_1, q_{01}, I, O, \delta_1, \lambda_1 \rangle$ is equivalent to another Mealy machine $\langle Q_2, q_{02}, I, O, \delta_2, \lambda_2 \rangle$ iff $\forall s \in I^* : \lambda_1(q_{01}, s) = \lambda_2(q_{02}, s)$. A counterexample to equivalence is thus an $s \in I^*$ such that $\lambda_1(q_{01}, s) \neq \lambda_2(q_{02}, s)$.

3.2 Active Automata Learning

We consider learning in the minimally adequate teacher (MAT) framework [2]. Algorithms in this framework infer models of black-box systems, also referred to as SULs, through interaction with a so-called teacher.

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 [2]. 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.

Learning Mealy Machines. Margaria et al. [18] and Niese [20] were one of 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 [25]. 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 [25] 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 conformance testing as implemented in LearnLib [14]. To summarise, a learning algorithm for Mealy machines relies on three operations:

reset: resets the SUL

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

equivalence query: conformance testing between SUL and hypothesis.

As shown in Fig. 1, 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.

Due to the incompleteness of testing, learned models may be incorrect. If, e.g., the W-method [6, 30] is used for testing, the learned model may be incorrect if assumptions placed on the maximum number of states of the SUL do not hold.

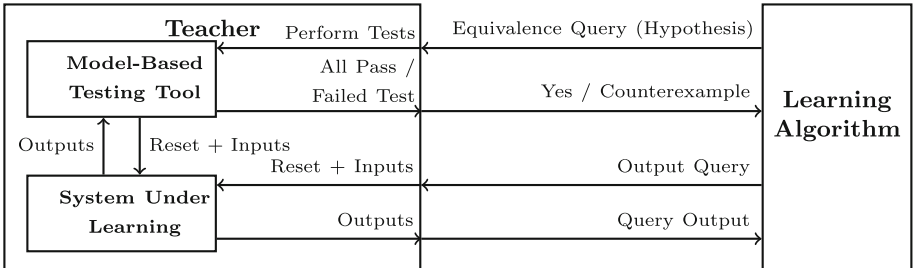


Fig. 1. The interaction between SUL, teacher and learning algorithm (based on [26]).

Algorithm 1. The test-case generation algorithm.

<pre> 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: $rS \leftarrow rSel(S_h) \triangleright (rS, rI)$ defines 9: $rI \leftarrow rSel(I) \triangleright$ a transition 10: $p \leftarrow path(state, rS)$ 11: if $p \neq None$ then 12: $rSteps \leftarrow rSeq(I, l_{infix})$ </pre>	<pre> 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 </pre>
---	--

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” [1]. Motivated by this, we developed a simple and 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. The test-case generation may start with a random walk through the model and then iterates two operations. First, a transition of the model is chosen randomly and a path leading to it is executed. If the transition is not reachable, another target transition is chosen. Second, another short random walk is executed. 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 $maxSteps$. Alternatively, it may also stop dependent on probabilities p_{retry} and p_{stop} . 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. The generation procedure uses three random functions. A function $coinFlip$ 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. $\forall e \in S: \mathbb{P}(rSel(S) = e) = \frac{1}{|S|}$. 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. Algorithm 1 realises the test-case generation based on \mathcal{M}_h . As additional inputs, it takes stopping parameters and $l_{\text{infix}} \in \mathbb{N}$, an upper bound on the number of 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 breadth-first exploration but other approaches are possible as long as they satisfy $\text{path}(s, s') = \text{None}$ iff $\nexists \bar{i} \in I^* : \delta(s, \bar{i}) = s'$ and $\text{path}(s, s') = \bar{i} \in I^*$ such that $\delta(s, \bar{i}) = s'$, where $\text{None} \notin I$ 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 three 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 . We greedily select test cases until either the upper bound n_{sel} is reached, all elements in C are covered, or we do not improve coverage. More formally:

<pre> 1: $T_{\text{sel}} \leftarrow \emptyset$ 2: while $T_{\text{sel}} < n_{\text{sel}} \wedge C \neq \emptyset$ do 3: $t_{\text{opt}} \leftarrow \operatorname{argmin}_{t \in T} C \setminus C_t$ 4: if $C \cap C_{t_{\text{opt}}} = \emptyset$ then 5: break \triangleright no improvement </pre>	<pre> 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 </pre>
---	--

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

4.3 Mutation-Based Selection

A particularly interesting selection criterion is mutation-based selection. The choice of this criterion is motivated by the fact that model-based mutation testing can effectively be combined with random testing [1]. Generally, in this fault-based test-case generation technique, known faults are injected into a model creating so-called mutants. Test cases are then generated which distinguish these mutants from the original model and thereby test for the corresponding faults.

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

Thus, in our case we alter the hypothesis \mathcal{M}_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 $C_t = \{\mathcal{M}_{\text{mut}} \in \mathcal{MS}_{\text{mut}} \mid \lambda_h(t) \neq \lambda_{\text{mut}}(t)\}$.

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 [15] and also finite-state machines [9]. 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 [22]. Thus, we developed a family of mutation operators, called *split-state* operators, directly addressing active automata learning.

Split-State Operator Family. There are different ways to process counterexamples in the MAT framework, such as by adding all prefixes to the used data structures [2]. An alternative technique due to Rivest and Schapire [23] 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. This knowledge is then integrated into the data structures.

Since it is an efficient form of processing counterexamples, adaptations of it have been used in other learning algorithms such as the TTT algorithm [13]. 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 1 illustrates this mutation operator.

Example 1. (Split State Mutation). A hypothesis produced by a learning algorithm may be of the form shown in Fig. 2a. Note that not all edges are shown in the figure and dashed edges represent sequences of transitions. The access sequences $\text{acc}(q_{h3})$ 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. 2b. In this case, the hypothesis incorrectly

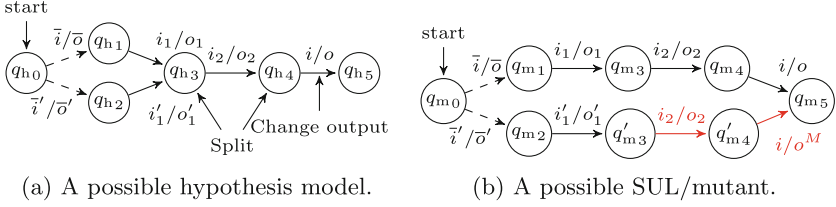


Fig. 2. Demonstration of split state.

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'_{m4} as indicated in Fig. 2b. State q_{h4} has to be split as well to introduce a distinguishing sequence of length 2 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}_{mut} , we either find an actual counterexample to equivalence between SUL and hypothesis or prove that the SUL does not implement \mathcal{M}_{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 [1].

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:

Max. number of sequences n_{acc} : an upper bound on the number of mutated access sequences leading to a single state.

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 $l < k$.

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 $\delta(s_{0h}, u') = \delta(s_{0h}, u)$, 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*.

Efficiency Considerations. While test-case generation can efficiently be implemented, mutation-based selection is 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 $|S| * n_{\text{acc}} * |I|^k$, this may become a bottleneck.

Consequently, cost reduction techniques for mutation [15] need to be considered. We reduce execution cost by avoiding the explicit creation of mutants.

Essentially only the difference to the hypothesis is stored and executed. Since this 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 [15]. 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.

In essence, the choice of the bound on the number of access sequences, the number of selected tests, the sample size, etc. needs 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. Additionally, the number of mutants may be reduced as follows.

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}_{mut} that $\lambda_h(t) = \lambda_{\text{sul}}(t)$ and $\lambda_h(t) \neq \lambda_{\text{mut}}(t)$ 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 [12] in the context of equivalence testing.

Adapting to learning algorithm: by considering the specifics of a learning algorithm, the number of access sequences could be reduced. For instance in discrimination-tree-based approaches [13], it would be possible to create mutants only for access sequences S stored in the discrimination tree and for their extensions $S \cdot I$. However, this has not been implemented yet.

5 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 *split state*. We compare these two techniques to alternatives in the literature: the partial W-method [11] and the random version of the approach discussed in [26] available at [19]. We refer to the latter as *random L & Y*. Note that this differs slightly from [10, 26] in which also non-randomised test, i.e. complete up to some bound, were generated.

We evaluate the different conformance testing methods based on two case studies from the domain of communication protocols. The examined systems are given in Table 1. This table includes the number of states and inputs of the true Mealy machine model and a short description of each system. Due to space limitations, we refer to other publications for in-depth descriptions.

Table 1. A short description of examined systems.

System	# States	# Inputs	Short description
TCP server (Ubuntu)	57	12	Models of TCP server/client implementations from three different vendors have been learned and analysed by Fiterău-Broştean et al. [10]. We simulated the server model of Ubuntu available at [29].
MQTT broker (emqtt [8])	18	9	Model of an MQTT [3] broker interacting with two clients. We discussed the learning setup in [28].

5.1 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, protocols, can be done fast (simply start a new session), whereas timeouts and quiescent behaviour cause long test durations [24, 28]. Note that we take previously learned models as SULs. Their simulation ensures fast test execution which enables a thorough evaluation.

To estimate the probability of learning the correct models, we performed each learning run 50 times and calculated the relative frequency of learning the correct model. In the following, we refer to such a repetition as a single experiment. Note that given the expected number of states of each system, we can efficiently determine whether the correct model has been learned, since learned models are minimal with respect to the number of states [2].

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. For learning with the partial W-method [11] we gradually increased the depth parameter implemented in LearnLib [14] 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 [14].

TCP – Ubuntu. The number of tests and steps required to reliably learn the Ubuntu TCP-server are given in Table 2. In order to perform these experiments, we generated 100,000 tests and selected the number of tests given in the first line

Table 2. Performance measurements for learning an Ubuntu TCP-server-model.

	Transition coverage	Split state	Partial W-method	Random L & Y
Bound on # equivalence tests/depth parameter	10,000	4,000	2	46,000
Mean # tests [equivalence]	12,498	4,786	793,939	71,454
Mean # steps [equivalence]	239,059	138,840	7,958,026	823,623
Mean # tests [membership]	9,633	10,017	13,962	11,445
Mean # steps [membership]	127,684	129,214	147,166	136,827

of Table 2 to perform each equivalence query. For the partial W-method this line includes the depth-parameter value. Note that the mean values of tests/steps represent the numbers summed over all rounds of learning (but averaged over 50 runs), while the bound on the number of tests applies to only a single round. The test-case generation with Algorithm 1 has been performed with parameters $maxSteps = 40$, $p_{retry} = 0.9$, $p_{stop} = 0.1$, and $l_{infix} = 6$. The chosen parameters for *split state* selection are $n_{acc} = 100$ (max. access sequences per state) and $k = 2$ (length of distinguishing sequence). Additionally, we performed mutant sampling by first reducing the number of mutants to one quarter of the original mutants and then to 10,000 if necessary.

We see in the table 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 less tests do not trade membership for equivalence tests. With this out of the way, we can concentrate on equivalence testing. We see that *split state* pays off in this experiment with *transition coverage* requiring 1.7 times as many steps. The average cost of test selection is 104 seconds for *split state* and 4 seconds for *transition coverage*. However, considering the large savings in actual test execution, *split state* performs better.

We also evaluated *random L & Y* with a middle sequence of expected length 4 (similar to [10]). For this setup, *random L & Y* requires significantly more steps and tests than both alternatives. There may be more suitable parameters, however, which would improve the performance of *random L & Y*. Nevertheless, the model structure of Ubuntu’s TCP-server seems to be beneficial for our approach.

All randomised approaches outperformed the partial W-method. In particular *split state* is able to reduce the number of test steps by a factor of 57. Taking the membership queries into account, the overall cost of learning is reduced by a factor of about 22. The relative gap between tests, a reduction by a factor of 166, is even larger. This is an advantage of our approach as we can flexibly control test length and thereby account for systems with expensive resets. Purely random testing is not a viable choice in this case. An experiment with 1,000,000 tests per equivalence query succeeded in learning correctly in only 4 of 50 runs.

MQTT – emqtt. The number of tests and steps required to reliably learn models of the emqtt broker are given in Table 3. In order to perform these experiments, we used largely the same setup and the same sampling strategy as for the TCP experiments, but generated only 10,000 tests as a basis for selection. Furthermore, we set $p_{\text{retry}} = 0.95$, $p_{\text{stop}} = 0.05$, $l_{\text{infix}} = 6$, $n_{\text{acc}} = 300$, and $k = 3$.

In Table 3, we see a similarly large improvement with respect to the partial W-method. The partial W-method requires about 52 times as many test steps as *split state*. Other than that, we see that the improvement of *split state* over *transition coverage* is not as drastic as for the Ubuntu TCP-server and testing with *random L & Y* also performs well. Figure 3 depicts the learning performance of the three different approaches and undirected random testing. It shows the dependency between the average number of equivalence-test steps and the estimated probability $\mathbb{P}_{\text{est}}(\text{correct})$ of learning the correct model. The graph shows that significantly more testing is required by random testing.

Table 3. Performance measurements for learning an emqtt-broker model.

	Transition coverage	Split state	Partial W-method	Random L & Y
Bound on # equivalence tests/depth parameter	275	125	2	1100
Mean # tests [equivalence]	345	182	72,308	1,679
Mean # steps [equivalence]	13,044	7,755	487,013	11,966
Mean # tests [membership]	1,592	1,623	1,808	1,683
Mean # steps [membership]	12,776	13,160	11,981	12,005

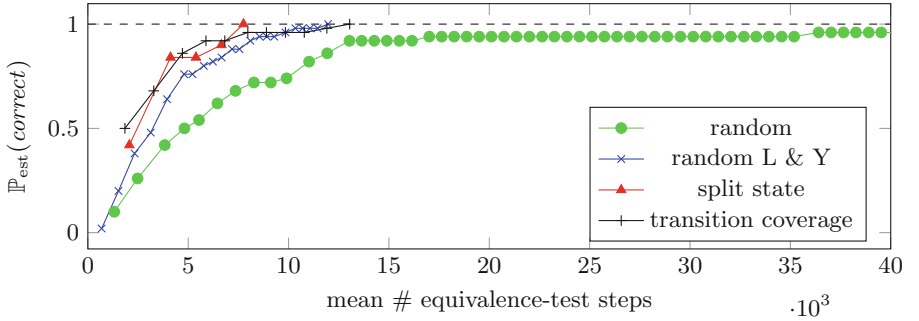


Fig. 3. Average number of equivalence-test steps required to reliably learn the correct emqtt-broker model.

Discussion and Threats to Validity. The results shown above suggest that *transition coverage* and especially *split state* perform well. However, the performance depends on the system structure. There are systems for which *transition*

coverage performs slightly better with regard to the required number of steps than *split state*, as the latter favours longer tests. In these cases, *split state* may simply add no value because *transition coverage* already performs well.

For the TCP-server case study, we report effectiveness superior to that of *random L & Y*. The performance of our technique depends on the concrete instantiation of more parameters than the performance of *random L & Y*. Finding suitable parameters is thus more difficult for our approach and relative performance gains may decrease for unsuitable choices. Additionally, *random L & Y* generates tests much more efficiently than *split state*. Split-state mutation analysis is only feasible for moderate-sized models, whereas *random L & Y* has successfully been applied for learning of a system with more than 3,000 states [26]. Mutation-based test-case selection would be hindered by the large number of mutants and tests forming the basis for selection – to our experience the number of tests should be increased with model size. More concretely, applying the technique to systems with significantly more than 100 states would likely not pay off. Aggressive mutant sampling would be necessary, rendering the mutation-based selection less effective. Without sampling, the decreased testing duration would not compensate for the cost of mutation analysis.

6 Conclusion

We presented a simple 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. First experiments showed 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 [11].

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. 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 [15] also hold for FSM mutation. Nevertheless, alternative mutant reduction techniques are not entirely exhausted. As indicated in Sect. 4.3, information stored by learning algorithms could help to reduce the number of mutants.

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 [5]. This would enable the application of more comprehensive fault models. Finally, alternatives to the simple greedy test-selection may also provide benefits.

Acknowledgment. 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 [19].

References

1. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W., Schlick, R., Tiran, S.: Killing strategies for model-based mutation testing. *Softw. Test. Verif. Reliab.* **25**(8), 716–748 (2015)
2. Angluin, D.: Learning regular sets from queries and counterexamples. *Inf. Comput.* **75**(2), 87–106 (1987)
3. Banks, A., Gupta, R. (eds.): MQTT Version 3.1.1. OASIS Standard, October 2014. Latest version: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>
4. 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.) FASE 2005. LNCS, vol. 3442, pp. 175–189. Springer, Heidelberg (2005). doi:[10.1007/978-3-540-31984-9_14](https://doi.org/10.1007/978-3-540-31984-9_14)
5. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. *Formal Asp. Comput.* **28**(2), 233–263 (2016)
6. Chow, T.S.: Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.* **4**(3), 178–187 (1978)
7. Combe, D., de la Higuera, C., Janodet, J.-C.: Zulu: an interactive learning competition. In: Yli-Jyrä, A., Kornai, A., Sakarovitch, J., Watson, B. (eds.) FSMNLP 2009. LNCS (LNAI), vol. 6062, pp. 139–146. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-14684-8_15](https://doi.org/10.1007/978-3-642-14684-8_15)
8. emqtt. <http://emqtt.io/>. Accessed 29 Nov 2016
9. Fabbri, S., Delamaro, M.E., Maldonado, J.C., Masiero, P.C.: Mutation analysis testing for finite state machines. In: ISSRE 1994, pp. 220–229. IEEE (1994)
10. Fiterău-Broștean, P., Janssen, R., Vaandrager, F.: Combining model learning and model checking to analyze TCP implementations. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 454–471. Springer, Cham (2016). doi:[10.1007/978-3-319-41540-6_25](https://doi.org/10.1007/978-3-319-41540-6_25)
11. 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)
12. Howar, F., Steffen, B., Merten, M.: From ZULU to RERS - lessons learned in the ZULU challenge. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6415, pp. 687–704. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-16558-0_55](https://doi.org/10.1007/978-3-642-16558-0_55)
13. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 307–322. Springer, Cham (2014). doi:[10.1007/978-3-319-11164-3_26](https://doi.org/10.1007/978-3-319-11164-3_26)
14. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 487–495. Springer, Cham (2015). doi:[10.1007/978-3-319-21690-4_32](https://doi.org/10.1007/978-3-319-21690-4_32)
15. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* **37**(5), 649–678 (2011)
16. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - a survey. *Proc. IEEE* **84**(8), 1090–1123 (1996)
17. Lee, D., Yannakakis, M.: Testing finite-state machines: state identification and verification. *IEEE Trans. Comput.* **43**(3), 306–320 (1994)
18. 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, pp. 95–100. IEEE Computer Society (2004)

19. Moerman, J.: Yannakakis - test-case generator. <https://gitlab.science.ru.nl/moerman/Yannakakis>. Accessed 30 Nov 2016
20. Niese, O.: An integrated approach to testing complex systems. Ph.D. thesis, Dortmund University of Technology (2003)
21. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Wu, J., Chanson, S.T., Gao, Q. (eds.) FORTE XII/PSTV XIX 1999. IFIP AICT, vol. 28, pp. 225–240. Springer, Boston (1999). doi:[10.1007/978-0-387-35578-8_13](https://doi.org/10.1007/978-0-387-35578-8_13)
22. 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)
23. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. *Inf. Comput.* **103**(2), 299–347 (1993)
24. de Ruiter, J., Poll, E.: Protocol state fuzzing of TLS implementations. In: *USENIX Security* 15, pp. 193–206. USENIX Association (2015)
25. Shahbaz, M., Groz, R.: Inferring Mealy machines. In: Cavalcanti, A., Dams, D.R. (eds.) *FM 2009. LNCS*, vol. 5850, pp. 207–222. Springer, Heidelberg (2009). doi:[10.1007/978-3-642-05089-3_14](https://doi.org/10.1007/978-3-642-05089-3_14)
26. Smeenk, W., Moerman, J., Vaandrager, F., Jansen, D.N.: Applying automata learning to embedded control software. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) *ICFEM 2015. LNCS*, vol. 9407, pp. 67–83. Springer, Cham (2015). doi:[10.1007/978-3-319-25423-4_5](https://doi.org/10.1007/978-3-319-25423-4_5)
27. Tappler, M.: mut-learn - randomised mutation-based equivalence testing. <https://github.com/mtappler/mut-learn>. Accessed 07 Dec 2016
28. Tappler, M., Aichernig, B.K., Bloem, R.: Model-based testing IoT communication via active automata learning. In: *ICST 2017*. IEEE Computer Society (2017)
29. TCP models. <https://gitlab.science.ru.nl/pfiteraubrosteian/tcp-learner/tree/cav-aec/models>. Accessed 14 Nov 2016
30. Vasilevskii, M.P.: Failure diagnosis of automata. *Cybernetics* **9**(4), 653–665 (1973)

NASA Formal Methods

9th International Symposium, NFM 2017, Moffett Field,

CA, USA, May 16-18, 2017, Proceedings

Barrett, C.; Davies, M.; Kahsai, T. (Eds.)

2017, XI, 436 p. 124 illus., Softcover

ISBN: 978-3-319-57287-1