Learning-Based Testing of an Industrial Measurement Device

Bernhard K. Aichernig[1], Christian Burghard[1,2](✉), and Robert Korošec[2]

[1] Institute of Software Technology, Graz University of Technology, Graz, Austria
{aichernig,burghard}@ist.tugraz.at
[2] AVL List GmbH, Graz, Austria
{christian.burghard,robert.korosec}@avl.com

Abstract. Active automata learning algorithms have gained increasing importance in the field of model-based system verification. For some classes of systems - especially deterministic systems, like Mealy machines, a variety of learning algorithm implementations is readily available. In this paper, we apply this technique to a measurement device from the automotive industry in order to systematically test its behaviour. However, our system under learning shows sparse non-deterministic behaviour, preventing the direct application of the available learning tools.

We propose an implementation of the active automata learning framework which masks this non-determinism. We repeat a previous model-based testing experiment with faulty devices and show that we can detect all injected faults. Most importantly, our technique was also able to find unknown bugs.

Keywords: Active learning · Automata learning · Model inference · Testing · Mutation analysis · Automotive case study · Testbed

1 Introduction

Due to the ever increasing complexity of industrial software and mechatronical systems, *model-based testing* (MBT) techniques have seen a popularity gain in the past two decades [13,32]. The practice of model-based testing involves the creation of a system model, using an appropriate abstraction of the *system under test* (SUT). From this model, test cases are automatically derived according to a specific test selection method. These test cases can then be executed on the SUT to either strengthen the trust in its conformance to the system model or to disprove said conformance. The sub-discipline of *model-based mutation testing* (MBMT) [3] deserves special mentioning. In MBMT, a set of *mutants*, i.e. faulty variants of the system model, is generated. Test cases are selected in order to maximize the number of mutants which can be distinguished from the original. Hence, MBMT is able to rule out the presence of specific faults and, under certain circumstances, is also able to subsume other common test-selection criteria [27].

However, the feasibility of MBT techniques strongly depends on the presence of adequate system models which are not always available. *Learning-based testing* (LBT) [6] is a complementary approach to the conformance testing approaches described above. Here, a learning algorithm infers a system model through interaction with a black-box system. This learned model can then be checked for the fulfilment of requirements [15,16] or for conformance to a reference model [31]. Due to its objective to explore the entire space of system behaviours without regards to a specification (restricted only by the chosen abstraction), LBT can be used for *fuzzing*, i.e. robustness testing [2,29]. The fact that LBT does not require the presence of system models is an important factor in its industrial application.

Industrial Use Case. The AVL List GmbH is the world's leading supplier of automotive test systems with over 9.500 employees worldwide. Its portfolio comprises, among other things, a wide variety of measurement devices for engine exhaust, fuel consumption, engine torque, etc. These measurement devices are usually arranged in the form of a testbed (e.g. for engines, powertrains or entire vehicles) and integrated into a *testbed automation system* which controls each device, e.g., over an Ethernet connection. In the past, we have developed two MBMT approaches to test this integration for a specific measurement device. Our first approach [4,23] used UML [28] to specify the system model. The second approach [10,11] used a *domain-specific* modelling language called *MDML*.

Summary and Contributions. In the work at hand, we present a case study regarding the application of a learning-based testing approach to the same exhaust measurement device. We further present our approach to mask sparse non-deterministic behaviour of this device to enable the use of off-the-shelf automata learning algorithms. We perform a *mutation analysis* on our LBT approach—i.e. we evaluate its fault detection capability relative to a set of mutated devices. Finally, we compare the results of the mutation analysis with those of our MBMT approaches which have been evaluated against the same set of mutants.

Our contributions are threefold: (1) Our case study provides further evidence that LBT can be successfully applied in industry and, most importantly, that it helps in finding bugs. (2) The mutation analysis shows that LBT finds more injected faults than our previous approaches with model-based test-case generation. To the best of our knowledge this is the first comparison of this kind. (3) We provide details of a mapper that speeds-up learning and masks occurrences of non-determinism.

Structure. Section 2 defines used formalisms and gives a background on active automata learning. In Sect. 3, we describe the measurement device under test and the various components of our learning setup. The learning results based on these implementations are presented in Sect. 4. We discuss related work in Sect. 5 and draw our conclusions in Sect. 6.

2 Preliminaries

2.1 Notational Conventions and Mealy Machines

Let $a \in A$ be a symbol from an alphabet A. We define *words* or *sequences* over this alphabet as $\bar{a} = [a_1, a_2, \ldots, a_n] \in A^*$ with the *empty word* ϵ and $A^+ = A^* \setminus \{\epsilon\}$. We lift symbols $a \in A$ to words $[a] \in A^*$. The set of words over A within a defined length range is written as $A^{\{m \ldots n\}} = \{\bar{a} \in A^* \mid m \leq |\bar{a}| \leq n\}$. We write the concatenation of words as $\bar{a} = \bar{a}_1 \cdot \bar{a}_2$. We write the n-fold self-concatenation of a word as $\bar{a}^n$ with $n \in \mathbb{N}_{>0}$ and $\bar{a}^1 = \bar{a}$, as well as $\bar{a}^*$ for undefined finite n. We define $last(\bar{a}) : A^+ \rightarrow A$ to return the last symbol a_n from a sequence of length n. We furthermore define the set of non-empty prefixes of a sequence $pre(\bar{a} \in A^+) = \{\bar{a}_p \in A^+ \mid \exists\, \bar{a}_s \in A^* : \bar{a} = \bar{a}_p \cdot \bar{a}_s\}$.

Definition 1 (Finite-state transducer (FST)). *A finite-state transducer is a 6-tuple $\langle Q, q_0, I, O, \delta, \lambda \rangle$ with a sets of states Q, an initial state $q_0 \in Q$, an input alphabet I, an output alphabet O, a state transition relation $\delta \subseteq Q \times I \times Q$ and an output relation $\lambda \subseteq Q \times I \times O$.*

If $\langle q, i, q' \rangle \in \delta$ and $\langle q, i, o \rangle \in \lambda$, we write $q \xrightarrow{i/o} q'$.

Definition 2 (Mealy machine). *A Mealy machine is an FST $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ with δ and λ restricted to functions: $\delta : Q \times I \rightarrow Q$ and $\lambda : Q \times I \rightarrow O$.*

If a Mealy machine $\mathcal{M}$ receives an input sequence $\bar{\imath} = [i_1, i_2, \cdots, i_n]$, such that $q_0 \xrightarrow{i_1/o_1} q_1 \xrightarrow{i_2/o_2} q_2 \cdots \xrightarrow{i_n/o_n} q_n$, producing an output sequence $\bar{o} = [o_1, o_2, \cdots o_n]$, we write $\mathcal{M}(\bar{\imath}) = \bar{o}$. Two Mealy machines $\mathcal{M}_1$ and $\mathcal{M}_2$ are called *equivalent* iff $\forall\, \bar{\imath} \in I^+ : \mathcal{M}_1(\bar{\imath}) = \mathcal{M}_2(\bar{\imath})$.

2.2 Active Automata Learning

Active automata learning [18] is the practice of inferring the internal behaviour of a black-box system by stimulating it with inputs and observing the produced outputs. One of the most important contributions in this field was made by Angluin [7] who proposed an algorithm called L* to infer regular sets of strings from a *minimally adequate teacher* (MAT). With slight modifications, the L* algorithm was later adapted to Mealy machines [26]. The MAT framework, depicted in Fig. 1, consists of a *learner* who implements the learning algorithm, and a *teacher* who usually encapsulates the *system under learning* (SUL). The learner is able to pose two types of questions to the teacher:

Output Query[1]**:** Here, the learner supplies an input word $\bar{\imath} \in I^+$ to the teacher who responds with an output word $\bar{o} \in O^+$ with $|\bar{o}| = |\bar{\imath}|$. Before an output query is executed, the teacher *resets* the SUL to a defined initial state. By posing output queries, the learner constructs a *hypothesis* $\mathcal{H}$ of the Mealy machine $\mathcal{S}$ which is

[1] Also known as *membership query* in literature.

assumed to be implemented in the SUL. Output queries may be handled by a component of the teacher called *output oracle* which resets the SUL, executes $\bar{\imath}$ symbol by symbol, records the SUL output and compiles it into $\bar{o}$. While this task is trivial by itself, its position in the data flow allows the output oracle to enact domain-specific performance-increasing caching operations as it is able to analyse each $\bar{\imath}$ before it is executed on the SUL.

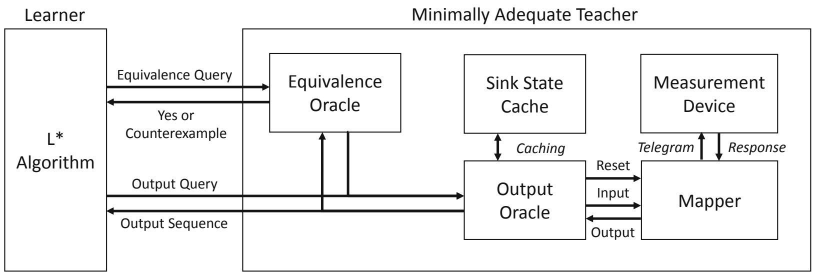

Fig. 1. *Minimally adequate teacher* (MAT) framework, including our specific implementation of the teacher.

Equivalence Query: When the learner has formed a hypothesis $\mathcal{H}$, it is forwarded to the teacher. If $\mathcal{H}$ is equivalent to $\mathcal{S}$, the teacher issues a positive response and the learning process concludes with $\mathcal{H}$ as its result. Otherwise, the teacher responds with a counterexample to equivalence $\langle \bar{\imath}, \bar{o} \rangle$ with $\mathcal{S}(\bar{\imath}) = \bar{o} \neq \mathcal{H}(\bar{\imath})$. The learner uses this counterexample and subsequent output queries to refine its hypothesis in a new round of learning. If the SUL is a black-box, equivalence queries cannot be executed directly. In this case, the teacher contains a component called *equivalence oracle*, which resolves an equivalence query into a series of output queries. In essence, the equivalence oracle performs a conformance test on the SUL with respect to the hypothesis [6]. As this approach to equivalence checking is subject to the inherent incompleteness of testing, it cannot guarantee that the learning result accurately captures the behaviour of a black-box SUL. In the work at hand, we use a guided random testing approach, described in Sect. 3.5.

2.3 Mapper

In most cases, an additional abstraction layer must be introduced between the SUL and the rest of the MAT framework, e.g., to reduce the size of an excessively large SUL state space or to augment the input or output alphabets [2]. For this purpose, Aarts et al. [1] have proposed a stateful *mapper* component to abstract the communication between the output oracle and the SUL.

Definition 3 (Mapper). *A mapper is an 8-tuple $\langle R, r_0, C_I, C_O, I, O, \Delta, \nabla\rangle$ with a set of states R, an initial state $r_0 \in R$, a concrete input alphabet C_I, a concrete output alphabet C_O, an abstract input alphabet I, an abstract output alphabet O, a state transition function $\Delta : R \times (C_I \cup C_O) \rightarrow R$ and an abstraction function $\nabla : (R \times C_I \rightarrow I) \cup (R \times C_O \rightarrow O)$.*

The output oracle operates on the mapper via the abstract alphabets I and O while the mapper uses the concrete alphabets C_I and C_O to interact with the SUL. If $\Delta(r, c) = r'$, we write $r \xrightarrow{c} r'$. We also use the mapper to reset the SUL to a defined initial state before performing an output query. Due to scope limitations and no direct relevance to our contribution, we cannot give a more detailed explanation of our reset operation in this work.

Definition 4 (Abstraction). *Let $\mathcal{S} = \langle Q, q_0, C_I, C_O, \delta, \lambda\rangle$ be an FST and let $\mathcal{A} = \langle R, r_0, C_I, C_O, I, O, \Delta, \nabla\rangle$ be a mapper. The abstraction of $\mathcal{S}$ via $\mathcal{A}$ is an FST $\alpha^{\mathcal{S}}_{\mathcal{A}} = \langle Q \times R, \langle q_o, r_o\rangle, I, O \cup \{\perp\}, \delta_\alpha, \lambda_\alpha\rangle$ so that δ_α and λ_α satisfy the following inference rules:*

$$\frac{q \xrightarrow{c_i/c_o} q', \quad r \xrightarrow{c_i} r' \xrightarrow{c_o} r'', \quad \nabla(r, c_i) = i, \quad \nabla(r', c_o) = o}{(\langle q, r\rangle, i, \langle q', r''\rangle) \in \delta_\alpha \ \wedge \ (\langle q, r\rangle, i, o) \in \lambda_\alpha}$$

$$\frac{\nexists\, c_i \in C_I : \nabla(r, c_i) = i}{(\langle q, r\rangle, i, \langle q, r\rangle) \in \delta_\alpha \ \wedge \ (\langle q, r\rangle, i, \perp) \in \lambda_\alpha}$$

If an abstract input $i \in I$ is received, a concrete input $c_i \in C_I$ is chosen non-deterministically, so that $\nabla(r, c_i) = i$. If no such c_i can be found, the mapper issues the output $\perp$. The abstraction of a concrete output $c_o \in C_O$ is straightforward since ∇ contains exactly one abstract output for each c_o and $r' \in R$.

In general, it is possible that $\alpha^{\mathcal{S}}_{\mathcal{A}}$ introduces additional non-determinism through $\mathcal{A}$ that has not been present in $\mathcal{S}$. Aarts et al. [1] have defined an abstraction $\alpha^{\mathcal{S}}_{\mathcal{A}}$ to be *adequate* for a Mealy machine $\mathcal{S}$ if it introduces no non-determinism and is itself perceived as a Mealy machine. Our definition of an abstraction is based on non-deterministic SULs in the form of FSTs rather than on deterministic SULs in the form of Mealy machines. As we will explain in detail in Sect. 3, we use $\alpha^{\mathcal{S}}_{\mathcal{A}}$ to *mask* non-determinism present in our SULs. We refer to an abstraction as *sufficiently adequate* relative to a given learning algorithm[2] if the algorithm is unable in practice to distinguish the abstraction from a Mealy machine.

3 Learning Setup

We implemented our learning setup in Java using LearnLib [20], a popular open-source library for automata learning. LearnLib is available under the Apache License 2.0 and provides a multitude of implementations for different components of the MAT framework, including learning algorithms, output oracles and

[2] Including the equivalence oracle, since both components produce output queries.

equivalence oracles. With the exception of the learning algorithm, we have created our own concrete implementations of these components, as described in the remainder of this section. Unfortunately, LearnLib does not currently support a learning algorithm for FSTs. Therefore, we will define a mapper which masks the non-deterministic behaviour of our SULs to such an extent that the induced abstraction approximates a Mealy machine well enough for L* and related algorithms to perform successful, stable and repeatable learning runs.

3.1 System Under Learning: The AVL489 Particle Counter

The *AVL Particle Counter*—or AVL489 for short—is an automotive measurement device designed to measure the particle concentration in engine exhaust by means of laser scattering [8]. The device can be operated in one of several discrete states corresponding to different activities, abbreviated by single letters:

$$D_X = \left\{ \begin{array}{l} P \textit{ (Pause)}, S \textit{ (Standby)}, U \textit{ (Purging)}, R \textit{ (ResponseCheck)}, \\ Z \textit{ (ZeroCheck)}, M \textit{ (Measurement)}, I \textit{ (Integ. Measurement)}, \\ L \textit{ (LeakageCheck)} \end{array} \right\}$$

The device can be remotely controlled by the testbed automation system via an Ethernet connection by means of the AK protocol [21]. The automation system always initiates the communication by sending an AK telegram, consisting of a 4-letter code. The device replies with an answer telegram, repeating back the code, followed by a data payload. Telegrams of the form `S***` initiate state transitions while those of the form `A***` are used to query specific device parameters or values. When transitioning between the operating states, the device may have to perform physical activities (e.g. opening/closing of valves) during which it may be unable to accept incoming commands. This behaviour is expressed by a second, orthogonal state dimension:

$$D_Y = \{R \textit{ (Ready)}, B \textit{ (Busy)}\}$$

In the event of an operating state change, the device may simultaneously switch to *Busy*. In this case, incoming commands are refused until the device autonomously switches back to *Ready* after a few seconds. In addition, the device can be operated either remotely, as previously explained, or manually, as represented by a third state dimension:

$$D_Z = \{R \textit{ (Remote)}, M \textit{ (Manual)}\}$$

The device can be switched to *Manual* control via the AK command `SMAN`. Usually, the only possible following interaction is a return to *Remote* via `SREM`. Neither are the commands `SMAN` and `SREM` refused during *Busy* phases, nor do they delay the return to *Ready*.

However, the *observable device state* $D = D_X \times D_Y \times D_Z$ does not uniquely identify the *actual device state* since the latter may also contain information dependent on its history or timing aspects. Generally, the properties described above are merely our expectations about the device's behaviour (based on common practice), which we intend to either confirm or refute.

We aim for an acceptable runtime of our learning setup in an industrial context. However, we also try to incorporate as little domain knowledge as possible into our learning setup, which limits our ability to perform domain-specific optimizations. Therefore, we based our learning approach on a few assumptions about our SUL which can be viewed as part of a *testing hypothesis* (cf. [17]):

Assumption 1. ***A:*** *All variants of the AVL489 measurement device examined in this work can be described as FSTs.* ***B:*** *The non-deterministic behaviours of each examined FST are limited to only a few isolated instances.*

Strictly speaking, we could regard each AK command as an input to our SUL and the data payload of its response as its output. For reasons described in Sect. 3.3, this is impractical. For each transition performed by our abstraction, we send multiple AK commands to the SUL. We use some of these commands to trigger a state transition and others to generate the output produced by the abstraction. Instead of substantially changing the relatively commonly used mapper definition [5,15] from Sect. 2.3, we will use the set of state setter actions $\mathcal{AK}_{\mathtt{S}}$, as well as the set of state retrieval actions $\mathcal{AK}_{\mathtt{A}}$ to describe our interactions with the device. Elements of both sets are composed of the command code and the data payload of the response. In the case of $\mathcal{AK}_{\mathtt{S}}$, the payload contains information about the success (*Ok*) or failure (*Error*)[3] of the action and in case of $\mathcal{AK}_{\mathtt{A}}$, it contains the observable device state D:

$$\mathcal{AK}_{\mathtt{S}} = \mathtt{S***} \times \{Ok, Error\}$$

$$\mathcal{AK}_{\mathtt{A}} = \{\mathtt{ASTA}\} \times D$$

Here, `S***` denotes the set of all S-telegrams. The `ASTA` telegram retrieves the observable device state D. Unless specified otherwise, we will use the symbols $x \in D_X$, $y \in D_Y$ and $z \in D_Z$ to refer to unspecified elements of their respective domains. We will abbreviate elements of D in the form xyz, e.g. PRR or UBz.

3.2 Masking Non-determinism with Sink States

Assumption 2. ***A:*** *If the SUL responds to an S-telegram with an Error message, the SUL state remains unchanged.* ***B:*** *If an* `ASTA` *response reveals no observable SUL state change after an S-telegram has been issued, no change of the actual SUL state has taken place.*

Based on Assumption 2, we introduced several optimizations to increase the performance and stability of the learning setup. In both cases of Assumption 2, we redirected the transition to a sink state. To cover case **B**, we introduce a dedicated output symbol *Inert* which is returned when the SUL performed a valid

[3] Simplified for the purpose of this work. In reality, the response to an S-command is either empty in the case of success or contains a specific error code.

self-transition. In both cases, *Error* or *Inert* are returned for all subsequent outputs until the SUL is reset. These behavioural augmentations are performed with the help of a mapper, which is described in detail in Sect. 3.3. Consequently, all self-transitions in the SUL are replaced with sink-transitions in the learned model. In a hypothetical post-processing step, these sink transitions can be easily turned back into self-transitions.

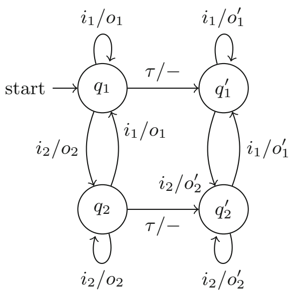

Fig. 2. Timing-induced non-determinism.

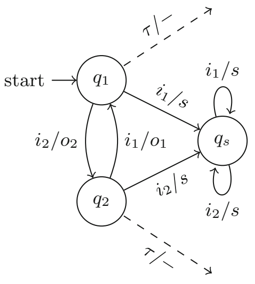

Fig. 3. Introducing a sink state q_s.

Our reasons for the introduction of sink states were twofold: on one hand, we attempted to prune the search space of the learning algorithm in order to increase its performance. On the other hand, the description of the measurement device as an FST yields instances of non-determinism like the one depicted in Fig. 2. These instances occur when a state q_1 exhibits both a self-transition as well as an internal transition $\tau/-$[4] to another state q_1' which is triggered after a certain amount of time has been spent in q_1. As the FST formalism does not account for the passing of time, the state q_1 is assumed to be unaltered after performing the self-transition. In reality however, each S-telegram sent to the measurement device takes several hundred milliseconds to process, effectively transitioning to a state with a reduced retention time until the timed transition is triggered. Therefore, a sequence of inputs i_1^* of sufficient length will eventually cause a transition to q_1'. The number of possible self-transitions before the time-out will most likely vary between experiments. If LearnLib encounters this behaviour repeatedly with different numbers of performed self-transitions, it will abort the learning process due to its inability to handle non-deterministic SULs.

This problem is mitigated by redirecting $\delta(q_1, i_1)$ to a sink state q_s, as depicted in Fig. 3. Once q_s has been reached, all transitions will output its specific sink label s. This modification was sufficient to keep the algorithm stable during hypothesis creation. However, in the case of loops of two or more transitions, the non-determinism cannot be masked by the introduction of a sink state. If the execution of a long *oscillating* sequence $[i_2, i_1]^*$ exceeds the time-out,

[4] In the notation for *labelled transition systems*, τ signifies an internal action which is not triggered via an external input. τ and inputs can be arbitrarily interleaved. "–" signifies *quiescence*—i.e. the absence of an output.

it will trigger a τ-transition in both Figs. 2 and 3. We will discuss our approach to this problem further in Sect. 3.5.

3.3 Mapper Implementation

Assumption 3. *The* `ASTA` *command always correctly retrieves the observable device state without influencing the actual device state.*

As explained in Sect. 3.1, the `ASTA` command retrieves the observable device state. According to Assumption 3, it behaves like a status message as described by Lee and Yannakakis [24]. Consequently, all interactions in $\mathcal{AK}_{\mathtt{S}}$ are independent of those in $\mathcal{AK}_{\mathtt{A}}$. To save further learning time, we directly incorporate Assumption 3 into the learning setup. For each individual step of the abstraction, we let the SUL perform both a state transition step, which is treated like an input, and a state retrieval step, which is treated like an output. The definition of the set of concrete input actions $C_I \subseteq \mathcal{AK}_{\mathtt{S}}{}^{\{0,1\}} \cup \mathcal{AK}_{\mathtt{A}}{}^{\{1\ldots n_p\}}$ allows for both S-telegrams as well as continuous polling for a timed device state change via `ASTA`. Hence, we may either send an S-telegram or actively wait and poll. With the maximum number of polls defined as $n_p = \lceil T_p \cdot f_p \rceil$, we choose the polling time-out $T_p = 25\,\mathrm{s}$ and the polling frequency $f_p = 10\,\mathrm{Hz}$. We choose the set of concrete output actions as $C_O \subseteq \mathcal{AK}_{\mathtt{A}}{}^{\{0,1\}}$. With these definitions in place, an exemplary transition sequence on the concrete SUL $\mathcal{S}$ could look like this:

$$q_0 \xrightarrow{\langle \mathtt{SPUL}, Ok\rangle / \langle \mathtt{ASTA}, UBR\rangle} q_1 \xrightarrow{\langle \mathtt{SPAU}, Error\rangle / \epsilon_{\mathrm{Error}}} q_1 \xrightarrow{\epsilon_{\mathtt{STBY}} / \epsilon_{\mathrm{Error}}} q_1$$

Note that, after $\mathcal{S}$ replies to `SPAU` with *Error*, only empty sequences of actions, i.e. no actions at all, are performed on $\mathcal{S}$. Simultaneously to this sequence performed by $\mathcal{S}$, we would like our abstraction $\alpha_{\mathcal{A}}^{\mathcal{S}}$ to perform

$$a_0 \xrightarrow{\mathtt{SPUL}/\,UBR} a_1 \xrightarrow{\mathtt{SPAU}/\,Error} a_{\mathrm{Error}} \xrightarrow{\mathtt{STBY}/\,Error} a_{\mathrm{Error}}$$

Table 1. Abstractions and state transitions performed by the mapper.

Mapper state $r \in R$	Abstract symbol $\nabla(r, c)$	Concrete symbol $c \in (C_I \cup C_O)$	Successor state $\Delta(r, c)$
Input abstraction			
$\langle d, \perp\rangle$	$i \in$ `S***`	$\langle i, Ok\rangle$	$\langle d, \perp\rangle$
$\langle d, \perp\rangle$	$i \in$ `S***`	$\langle i, Error\rangle$	$\langle d, Error\rangle$
$\langle d, \perp\rangle$	*Wait*	$\overline{poll}(d)$	$\langle d, \perp\rangle$
$\langle d, s \in \{Error,\ Inert\}\rangle$	$i \in I$	ϵ_i	$\langle d, s\rangle$
Output abstraction			
$\langle d, \perp\rangle$	d' with $d' \neq d$	$\langle \mathtt{ASTA}, d'\rangle$	$\langle d', \perp\rangle$
$\langle d, \perp\rangle$	*Inert*	$\langle \mathtt{ASTA}, d\rangle$	$\langle d, Inert\rangle$
$\langle d, s \in \{Error,\ Inert\}\rangle$	s	ϵ_s	$\langle d, s\rangle$

Table 2. State transition and output behaviour of the sink cache.

Output $\lambda_K(K, \bar{\imath})$	Successor State $\delta_K(K, \bar{\imath})$	Condition
$\alpha_{\mathcal{A}}^{\mathcal{S}}(\bar{\imath})$	K	$\Lambda(K, \bar{\imath}) = \bot \wedge \mathit{last}(\alpha_{\mathcal{A}}^{\mathcal{S}}(\bar{\imath})) \notin S$
$\alpha_{\mathcal{A}}^{\mathcal{S}}(\bar{\imath})$	$K \cup \{\langle \bar{\imath}, \alpha_{\mathcal{A}}^{\mathcal{S}}(\bar{\imath}) \rangle\}$	$\Lambda(K, \bar{\imath}) = \bot \wedge \mathit{last}(\alpha_{\mathcal{A}}^{\mathcal{S}}(\bar{\imath})) \in S$
$\bar{o}_p \cdot \mathit{last}(\bar{o}_p)^n$	K	$\Lambda(K, \bar{\imath}) = \bar{o}_p \in O^+ \wedge n = \lvert\bar{\imath}\rvert - \lvert\bar{o}_p\rvert$

Here, `SPAU` triggers a transition to a sink state a_{Error} which enforces the output symbol *Error* for all subsequent transitions. As suggested by this example, we choose the abstract input alphabet $I = \texttt{S***} \cup \{\mathit{Wait}\}$ as the set of S-telegrams plus an additional *Wait* symbol to abstract the polling for timed transitions. We choose our abstract output alphabet $O \subseteq D \cup S$ with the set of sink labels $S = \{\mathit{Error}, \mathit{Inert}\}$. To achieve this behaviour, we designed a mapper $\mathcal{A} = \langle R, r_0, C_I, C_O, I, O, \Delta, \nabla \rangle$. The mapper states $R \subseteq D \times (S \cup \{\bot\})$ comprise the last observed device state and either a sink label or, alternatively, $\bot$. We define the state transition function Δ and the abstraction function ∇ as per Table 1. We define the polling sequence for timed transitions as $\overline{\mathit{poll}}(d \in D) = [\langle \texttt{ASTA}, d_1 \rangle \ldots \langle \texttt{ASTA}, d_n \rangle]$ with $(n \leq n_p \wedge d_n \neq d \wedge \forall j \in \{1 \ldots n-1\} : d_j = d) \vee n = n_p$.

3.4 Output Oracle Implementation

As previously mentioned in Sect. 2.2, an output oracle is ideally suited to perform caching operations on output queries to increase the performance of the learning setup[5]. In particular, we are interested in input words containing a prefix which in the past has been observed to lead to a sink state. We now know beforehand that each input word containing the same prefix will inevitably lead to the same sink state which allows us to emulate the output word without performing expensive SUL interactions. Therefore, we let our output oracle implement a sink cache, as it was previously done by Stone et al. [30]. This technique is very similar to the *prefix closure filter* for Discrete Finite Automata (DFAs) described by Margaria et al. [25] as well as Hungar et al. [19]. Emulating output words led to a substantial performance increase of our learning setup which allowed us to perform learning runs of the AVL489 device overnight.

Definition 5 (Sink Cache). *Let $\alpha_{\mathcal{A}}^{\mathcal{S}} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ be a Mealy machine, and let $S \subset O$ be a set of sink labels. A sink cache is a Mealy machine $\langle \mathcal{K}, \emptyset, I^+, O^+, \delta_K, \lambda_K \rangle$ with $\mathcal{K} \subset \mathcal{P}(I^+ \times O^+)$ being the set of possible cache states. δ_K and λ_K are defined in Table 2 utilizing the cache lookup function $\Lambda : \mathcal{K} \times I^+ \to O^+ \cup \{\bot\}$:*

$$\Lambda(K, \bar{\imath}) = \begin{cases} \bar{o}_p & \textbf{\textit{if}}\ \exists!\ \bar{\imath}_p \in \mathit{pre}(\bar{\imath}), \bar{o}_p \in O^+ : \langle \bar{\imath}_p, \bar{o}_p \rangle \in K \\ \bot & \textbf{\textit{otherwise}} \end{cases}$$

[5] In addition to the caching operations defined above, we also filter redundant output queries, as described by Margaria et al. [25].

3.5 Equivalence Oracle Implementation

Our equivalence oracle performs a random walk through the transitions of the hypothesis and compares the transition outputs with those produced by the abstract SUL. However, we were unable to use the standard random-walk oracle implementation of LearnLib due to two additional requirements: (1) We want to end the current sequence and start a new one when and only when the random walk chooses a transition to a sink state. Otherwise we would either waste oracle steps while being stuck in a sink state or squander the chance of testing a nearby sink transition before ending the sequence. (2) We need the ability to terminate a sequence early if the strict avoidance of sink transitions forces the random walk into an oscillating sequence. In essence, we need to *weaken* our learning setup for $\alpha_{\mathcal{A}}^{\mathcal{S}}$ to be sufficiently adequate. We mitigate this problem through an empirically chosen approach which we will describe informally due to scope limitations: After reaching a state q within the hypothesis, we assign probabilities $p_i \in [0, 1]$ to each outgoing transition such that their sum equals 1. If the transition leads to a sink state, p_i is diminished relative to the other transition probabilities by a fixed factor. We also make p_i indirectly proportional to the number of times transition i has been chosen more often than the least-chosen transition originating in q. If the structure of the hypothesis funnels the random exploration into an oscillating sequence, this measure re-introduces an amount of fairness into the random choice and will eventually allow a sink transition to overrule the oscillating transitions. Dependent on the hypothesis, our equivalence oracle yielded a mean sequence length of around 30 ± 3 symbols and, within 3000 steps performed in total, a longest occurring sequence of about 100 ± 20 symbols.

3.6 Testbed Simulation Model

The AVL Test Center is tasked with testing the integration of various automotive measurement devices into the testbed automation system. As this integration needs to be tested for a multitude of different measurement device combinations and device firmware versions, it is infeasible to have all the actual devices present at the Test Center. Instead, the test engineers use a *Testbed Simulator* (TBSimu) which is able to emulate the AK interfaces of many different measurement devices over Ethernet. For our experiments, the test engineers provided us with a custom-made TBSimu model of AVL489 which can be configured to exhibit one of sixteen different implementation faults. This setup was previously used to evaluate the effectivity of both the UML-based and MDML-based MBMT approaches. Therefore, it stood to reason to use the same model for the evaluation of our LBT approach. Using the terminology of mutation testing, we will refer to the faulty configurations of the simulation model as *SUL mutants.* We named the respective mutants $\mathcal{S}_1$ to $\mathcal{S}_{16}$, as well as $\mathcal{S}_0$ for the error-free SUL. We refer to their respective learned models as $\mathcal{M}_0 - \mathcal{M}_{16}$. An overview of the individual SUL mutants is given in Table 3.

4 Results

4.1 Learned SUL Mutant Models

We ran the learning setup on each of the SUL mutants, as well as on $\mathcal{S}_0$. We then converted each learned model into a human-readable transition table. These transition tables were manually analysed and compared to the mutant specifications. Without exception, all learned mutant models showed differences to $\mathcal{M}_0$, thereby revealing the presence of errors. Some of the learned models showed partial deviations from the mutant specifications in Table 3 - either due to implementation errors or imprecise communication of the specification. Two of the learned models (specifically, $\mathcal{M}_{10}$ and $\mathcal{M}_{11}$) suggested the presence of highly unexpected behavioural anomalies within their respective SULs. These anomalies were later confirmed by manual experiments. All models, including $\mathcal{M}_0$, showed one common fault respective to our expectations from Sect. 3.1: Although the SULs were supposed to reject all commands except `SMAN` and `SREM` while in *Manual* mode, all *Manual* states accepted the same inputs as their *Remote* counterparts and reacted analogously. This effectively turned the *Manual* state space into a copy of the *Remote* state space.

Table 3. An overview of the SUL mutants, as well as mutant detection results of the LBT approach and both MBMT approaches.

ID	Mutant description	UML	MDML	LBT
1	`SMAN` (*Manual*) disabled in *Measurement*	✓	✓	✓
2	`SMAN` (*Manual*) disabled in *Integ. Measurement*	✓	✓	✓
3	`SMAN` (*Manual*) disabled in *Purging*	✓	✓	✓
4	No *Busy* phase when changing to *Pause*	✓	✓	✓
5	No *Busy* phase when changing to *Standby*	✓	✓	✓
6	No *Busy* phase when changing to *LeakageCheck*	✓	✓	✓
7	`SREM` (*Remote*) disabled in *ZeroCheck*	✗	✗	✓
8	`SREM` (*Remote*) disabled in *Purging*	✓	✗	✓
9	`SREM` (*Remote*) disabled in *LeakageCheck*	✗	✗	✓
10	Duration of *Busy* phases divided in half	✓	✓	(✓)
11	Duration of *Busy* phases doubled	✓	✓	✓
12	`SMGA` (*Measurement*) disabled	✓	✓	✓
13	`SINT` (*Integ. Measurement*) disabled	✓	✓	✓
14	`SPUL` (*Purging*) disabled	✓	✓	✓
15	`SNGA` (*ZeroCheck*) disabled	✓	✓	✓
16	Additional *Busy* phase when re-entering *Pause*	✓	✗	✓
Mutation Score (mutant detection ratio):		87.5%	75.0%	100.0%

Disregarding this common error, the models $\mathcal{M}_1 - \mathcal{M}_3$, $\mathcal{M}_7 - \mathcal{M}_9$ and $\mathcal{M}_{12} - \mathcal{M}_{15}$ were consistent with their respective mutant specifications. $\mathcal{M}_4 - \mathcal{M}_6$ showed the specified mutation for transitions triggered by $i \in \texttt{S***}$, but not for $i = \mathit{Wait}$. In contrast to the specification, $\mathcal{M}_{16}$ exhibited a general mutation of all transitions $q \xrightarrow{i/PRz} q'$ to $q \xrightarrow{i/PBz} q''$. For $\mathcal{M}_{10}$ and $\mathcal{M}_{11}$, the analysis of the learning results was more complicated. We expected the algorithm to miss $\mathcal{S}_{10}$ due to its shorter *Busy* phases. We also expected $\mathcal{M}_{11}$ to have the latter transition of $q \xrightarrow{\texttt{SPUL}/UBz} q' \xrightarrow{Wait/PBz} q''$ changed to $q' \xrightarrow{Wait/Inert} q_{\text{Inert}}$ due to the *Wait*-transition exceeding our polling time-out T_p. Neither expectation was reflected in the learning results. Instead, a closer examination of the learned models and their respective SULs revealed an implementation error of the aforementioned timed transition. In $\mathcal{M}_{10}$, the transition was changed in a way that decoupled the timed transition from *Purging* to *Pause* from the *Busy* time-out, resulting in $q \xrightarrow{\texttt{SPUL}/UBz} q' \xrightarrow{Wait/URz} q'' \xrightarrow{Wait/PBz} q'''$. Had $\mathcal{S}_{10}$ been implemented correctly, our LBT approach would most likely have missed it. In $\mathcal{S}_{11}$, the state *Purging* did not exhibit a lengthened *Busy* phase when left via *Wait* but instead passed it on to *Pause* when prematurely left via `SPAU`. This caused a split of *Pause* into two states with different *Wait*-transitions (see Fig. 4).

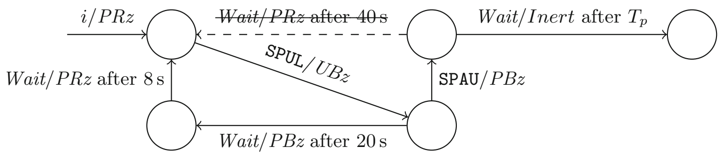

Fig. 4. Split of state $\langle \mathit{Pause}, \mathit{Busy}, z \rangle$ as captured by $\mathcal{M}_{11}$.

4.2 Discussion and Comparison to MBMT Approaches

In Table 4, we give an overview over a selection of learned mutant models with their number of states, the total number of non-redundant output queries, the number of queries executed on the SUL, the percentage of queries filtered by the sink cache, the same quantities regarding the individual steps and the total learning time. The percentage of filtered queries is consistent with the results of another case study utilizing a prefix-closure filter [25] as well as one on a version of L* optimized for prefix-closed automata [9]. With only few instances of hidden information, the device state is highly observable via the `ASTA` command. Therefore, the learning algorithm managed to capture their behaviour correctly in its initial hypothesis, without any counterexamples provided by the equivalence oracle. The only exception to this was $\mathcal{S}_{10}$ which contained states that could only be distinguished by at least two subsequent inputs. Hence, $\mathcal{M}_{10}$ was the only model to require counterexample processing, thereby yielding a high learning time despite the increased throughput of $\mathcal{S}_{10}$. In its particular case, the equivalence oracle would need to find the sequence [`STBY`, *Wait*, `SPUL`, *Wait*, *Wait*] which constitutes the shortest possible counterexample to the initial hypothesis produced by L*. However, our equivalence oracle is not guaranteed to produce the shortest counterexample. Instead, it repeatably produced longer and more indirect variants of the same counterexample.

Table 4. Comparison of selected learned mutant models in terms of state space and learning effort: C (first hypothesis correct), F (faster SUL), S (slower SUL).

ID	States	Queries			Steps			Time [hh:mm:ss]	Comment
		Total	SUL	Filtered	Total	SUL	Filtered		
0	32	4737	1593	66.4%	27972	11532	58.8%	09:52:30	C
4	32	4735	1615	65.9%	27972	11640	58.4%	09:48:01	C
6	30	4464	1512	66.1%	26100	10932	58.1%	09:51:11	C
10	36	9160	2849	68.9%	100255	34433	65.7%	15:35:40	F
11	34	5075	1667	67.2%	29268	11640	60.2%	17:15:15	C, S
12	22	3298	970	70.6%	17748	7164	59.6%	06:43:17	C
14	28	4178	1370	67.2%	25380	10500	58.6%	07:12:50	C

The LBT approach managed to distinguish all SUL mutants from the original, unlike both the UML- and MDML-based MBMT approaches. The UML-based approach suffered from performance bottlenecks in the test case generation algorithm and missed $\mathcal{S}_7$ and $\mathcal{S}_9$ due to its restricted exploration depth [4]. We show the combined results from all three test suites examined in the UML case study in Table 3. The MDML-based approach greatly improved upon the performance of the UML-based approach. However, at the time of its evaluation [10], it lacked an appropriate model *de-factoring* step which led to a detrimental relation between test model conciseness and test suite quality: If the system model was encoded very efficiently, small model mutations on the syntactic level could produce major changes on the semantic level. These coarse mutations led to the generation of weaker test suites. The generation of negative tests (i.e. testing the refusal of commands) was deliberately foregone in that case study to demonstrate the ability to generate useful test suites from underspecified MDML models. As a result, the common fault regarding the *Manual* mode was missed by the MDML-based approach. In contrast, automata learning techniques are only biased by the extent of the provided input alphabet and possible assumptions implemented in the mapper. The UML-based approach was evaluated against a different SUL implementation than both the MDML and LBT approaches, which may not have contained this fault. The above factors highlight the importance of careful mutant selection in MBMT approaches, as well as the relevance of *fuzzing* aspects in LBT approaches. While our learning runs take significantly more time than the execution of the generated test suites (29–96 min for UML and 12–15 min for MDML), they are still short enough to be performed overnight.

The AVL Test Center currently uses MDML models to generate test suites for measurement devices. While the use of MDML has drastically reduced the effort required for the creation and maintenance of test suites, it still requires an initial modelling effort which, in the absence of complete specifications, may involve a certain amount of guesswork. LBT on the other hand requires no a-priori modelling effort. Only the definition of an adequate abstraction is required, which can be re-used for similar SULs that share the same interface. It is also worth noting that our LBT approach is based on a significantly smaller technology stack than MDML-based test case generation, which comprised a substantial tool chain [10]. These factors make LBT attractive for our industrial use case.

5 Related Work

The most relevant related work for our use case is that of Stone et al. [30], who learned a common handshake protocol for Wi-Fi routers and had to deal with non-determinism caused by a lossy communication medium which manifested in time-out violations and message retransmissions. In contrast to our pre-emptive usage of sink states, they redirect the learning algorithm to a sink state after a non-deterministic re-transmission has already occurred. The non-determinisms were later discarded from the learning results through the repetition of output queries and a majority vote. Previously, Fiterău-Broştean et al. have employed a number of measures to deal with the same problem when learning the TCP protocol [14,15]. The authors masked time-outs by limiting the length of their output queries. De Ruiter and Poll [29] have utilized LBT to analyse TLS implementations. In their application domain, a sink state implicitly occurs when a TLS connection is closed. As their equivalence oracle, they used a modified variant of the W-Method [12] which filters prefixes navigating to this sink state. In their work on the inference of the Session Initiation Protocol [1], Aarts et al. introduced the technique of abstracting large SUL alphabets with the help of a mapper. All of the above approaches are either fully or partially based on LearnLib [20]. Both Hungar et al. [19] and Margaria et al. [25] present a number of query filtering techniques based on domain knowledge and evaluate different configurations thereof in their respective case studies. Berg et al. [9] have examined the scalability of L* on DFAs and modified the learning algorithm to perform prefix-closure filtering.

6 Conclusion and Outlook

We have created a learning setup based on LearnLib which offers a sufficiently adequate abstraction to the AVL489 measurement device and which can be executed within reasonable time. Our experiments have shown that the setup masks non-deterministic behaviour of our SUL reliably enough to enable the use of L* in AVLs industrial environment. We are confident that this method can be used to learn other systems which exhibit similar sparse (timing-induced) non-determinism. We have further shown that our learning setup is sensitive enough to uncover not only specified faults, but also unforeseen implementation errors within a measurement device simulation model. In contrast to the previously studied MBMT approaches, our LBT approach requires no initial modelling effort which makes it attractive to our industrial setting.

However, our masking of non-deterministic behaviour is still imperfect, since a hypothetical learning algorithm could issue membership queries containing sufficiently long oscillating sequences to circumvent our masking mechanism. In principle, our equivalence oracle is also able to produce oscillating sequences although we took measures to reduce their probability of occurrence.

In fact, the non-deterministic behaviour of our SUL is rooted in its description as an FST which lacks the concept of time. It is possible that AVL489 could be described as a *Mealy machine with timers*, as introduced by Jonsson and Vandraager [22] who also proposed a suitable learning algorithm. However, the absence of a respective implementation is still an issue. From an industrial point of view, LBT would lend itself to the automatic enhancement and maintenance of MDML models for regression testing. Alternatively, MDML models containing mere fragments of a device's behaviour could be used to specify requirements for the validation of learned device models.

Acknowledgements. Part of this work was supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". We thank the four anonymous reviewers and Martin Tappler for their valuable feedback.

References

1. Aarts, F., Jonsson, B., Uijen, J.: Generating models of infinite-state communication protocols using regular inference with abstraction. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 188–204. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16573-3_14
2. Aarts, F., de Ruiter, J., Poll, E.: Formal models of bank cards for free. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, 18–22 March 2013, pp. 461–468. IEEE (2013). https://doi.org/10.1109/ICSTW.2013.60
3. Aichernig, B.K.: Model-based mutation testing of reactive systems. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 23–36. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39698-4_2
4. Aichernig, B.K., et al.: Model-based mutation testing of an industrial measurement device. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 1–19. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09099-3_1
5. Aichernig, B.K., Bloem, R., Ebrahimi, M., Tappler, M., Winter, J.: Automata learning for symbolic execution. In: 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, 30 October–2 November 2018. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8602991

6. 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. LNCS, vol. 11026, pp. 74–100. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96562-8_3
7. 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
8. AVL List GmbH: AVL Particle Counter - Product Guide, AT2858E, Rev. 08 (2013)
9. Berg, T., Jonsson, B., Leucker, M., Saksena, M.: Insights to Angluin's learning. Electron. Notes Theoret. Comput. Sci. **118**, 3–18 (2005). https://doi.org/10.1016/j.entcs.2004.12.015
10. Burghard, C.: Model-based testing of measurement devices using a domain-specific modelling language. Master's thesis, Graz University of Technology, Institute of Software Technology (2018). http://truconf.ist.tugraz.at/wp-content/uploads/2018/04/MastersThesis_ChristianBurghard.pdf
11. Burghard, C., Stieglbauer, G., Korošec, R.: Introducing MDML - a domain-specific modelling language for automotive measurement devices. In: Joint Proceedings of the International Workshop on Quality Assurance in Computer Vision and the International Workshop on Digital Eco-Systems Co-Located with the 28th International Conference on Testing Software and Systems (ICTSS), pp. 28–31. CEUR-WS.org (2016). http://ceur-ws.org/Vol-1711/paperDECOSYS1.pdf
12. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Software Eng. **4**(3), 178–187 (1978). https://doi.org/10.1109/TSE.1978.231496
13. Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. In: Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007, pp. 31–36. ACM (2007). https://dl.acm.org/citation.cfm?id=1353681
14. Fiterău-Broştean, P., Janssen, R., Vaandrager, F.: Learning fragments of the TCP network protocol. In: Lang, F., Flammini, F. (eds.) FMICS 2014. LNCS, vol. 8718, pp. 78–93. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10702-8_6
15. Fiterău-Broştean, 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). https://doi.org/10.1007/978-3-319-41540-6_25
16. Fiterau-Brostean, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F.W., Verleg, P.: Model learning and model checking of SSH implementations. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, 10–14 July 2017, pp. 142–151. ACM (2017). https://doi.org/10.1145/3092282.3092289
17. Gaudel, M.-C.: Testing can be formal, too. In: Mosses, P.D., Nielsen, M., Schwartzbach, M.I. (eds.) CAAP 1995. LNCS, vol. 915, pp. 82–96. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59293-8_188
18. Howar, F., Steffen, B.: Active automata learning in practice. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits. LNCS, vol. 11026, pp. 123–148. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96562-8_5
19. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 315–327. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_31

20. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib: A framework for active automata learning. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 487–495. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_32
21. Jogun, K.: A universal interface for the integration of emissions testing equipment into engine testing automation systems: the VDA-AK SAMT-interface. Technical report, SAE Technical Paper (1994). https://doi.org/10.4271/940965
22. Jonsson, B., Vaandrager, F.W.: Learning Mealy machines with timers (2018). Preprint at http://www.sws.cs.ru.nl/publications/papers/fvaan/MMT/
23. Krenn, W., Schlick, R., Aichernig, B.K.: Mapping UML to labeled transition systems for test-case generation. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 186–207. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17071-3_10
24. 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
25. Margaria, T., Raffelt, H., Steffen, B.: Knowledge-based relevance filtering for efficient system-level test-based model generation. Innovations Syst. Softw. Eng. **1**(2), 147–156 (2005). https://doi.org/10.1007/s11334-005-0016-y
26. Niese, O.: An integrated approach to testing complex systems. Ph.D. thesis, Technical University of Dortmund, Germany (2003). https://doi.org/10.17877/DE290R-14871
27. Offutt, A.J., Voas, J.M.: Subsumption of condition coverage techniques by mutation testing. Technical report, George Madison University, Fairfax, VA, USA (1996). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.8904&rep=rep1&type=pdf
28. OMG: OMG Unified Modeling Language (OMG UML), Version 2.5.1. Object Management Group, August 2017. http://www.omg.org/spec/UML/2.5.1
29. de Ruiter, J., Poll, E.: Protocol state fuzzing of TLS implementations. In: 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, 12–14 August 2015, pp. 193–206. USENIX Association (2015). https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter
30. McMahon Stone, C., Chothia, T., de Ruiter, J.: Extending automated protocol state learning for the 802.11 4-way handshake. In: Lopez, J., Zhou, J., Soriano, M. (eds.) ESORICS 2018. LNCS, vol. 11098, pp. 325–345. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99073-6_16
31. 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, 13–17 March 2017, pp. 276–287. IEEE (2017). https://doi.org/10.1109/ICST.2017.32
32. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verification Reliab. **22**(5), 297–312 (2012). https://doi.org/10.1002/stvr.456