Quick Bug Detection through Black-Box Checking

A Systematic Evaluation

Bram Pellen
Radboud University
Nijmegen, Netherlands
bram.pellen@ru.nl

María Belén Rodríguez
University of Twente
Enschede, Netherlands
mariabelen.rodriguez@utwente.nl

Frits Vaandrager
Radboud University
Nijmegen, Netherlands
F.Vaandrager@cs.ru.nl

Petra van den Bos
University of Twente
Enschede, Netherlands
p.vandenbos@utwente.nl

Abstract

Combinations of active automata learning, model-based testing and model checking have been successfully used in numerous applications, e.g., for spotting bugs in implementations of major network protocols and to support refactoring of embedded controllers. However, in the large majority of these applications, model checking is only used at the very end, when no counterexample can be found anymore for the latest hypothesis model. This contrasts with the original proposal of black-box checking (BBC) by Peled, Vardi & Yannakakis, which applies model checking for *all* hypotheses, also the intermediate ones. In this article, we present the first systematic evaluation of the ability of BBC to find bugs quickly, based on 77 benchmark models from real protocol implementations and controllers for which specifications of safety properties are available. Our main finding are: (a) In cases where the full model can be learned, BBC detects violations of the specifications with just 3% of the queries needed by an approach in which model checking is only used for the full model. (b) Even when the full model cannot be learned, BBC is still able to detect many violations of the specification. In particular, BBC manages to detect 96% of the safety property violations in the challenging RERS 2019 industrial LTL benchmarks. (c) Our results also confirm that BBC is way more effective than existing model-based testing algorithms in finding deep bugs in implementations.

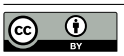
CCS Concepts

• **Software and its engineering** → **Formal methods**; **Software verification and validation**.

Keywords

Black-box checking, Active automata learning, Model learning, Model checking, Model-based testing, Runtime monitoring

ACM Reference Format:
Bram Pellen, María Belén Rodríguez, Frits Vaandrager, and Petra van den Bos. 2026. Quick Bug Detection through Black-Box Checking: A Systematic Evaluation. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3834347



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834347

1 Introduction

Testing whether a software system conforms to a high-level specification is challenging without access to source code or a detailed, low-level model. Without guidance for the selection of tests inputs, testers typically resort to random test generation algorithms, possibly in combination with user-provided input grammars and evolutionary mechanisms [28, 65]. However, such methods can easily fail to expose software bugs whose occurrence depends upon specific sequences of input events.

In the literature, some powerful learning-based testing techniques have been proposed that address these limitations by combining testing and learning [4, 38, 39, 43, 44, 49]. These techniques are based on ideas of [8, 61] about the duality between inductive inference and conformance testing: where inductive inference aims to construct a hypothesis model based on the outcomes of a finite number of experiments, the goal of conformance testing is to find a counterexample for this hypothesis by performing experiments. The central idea of learning-based testing is to create a feedback loop in which counterexamples found during testing help to make the learned models more accurate, whereas the learned models drive the test generation to obtain increasingly powerful test suites.

Learning-based testing has been successfully used in numerous applications, e.g., for spotting security vulnerabilities and bugs in implementations of major network protocols [13, 17–22] and to support refactoring of embedded controllers [50, 51, 62]. We refer to [2, 12, 30, 56] for surveys and further references. As a result, learning-based testing has become a standard tool in the toolbox of software engineers who are trying to find deep bugs in protocol implementations or state-machine-based embedded controllers.

In their seminal paper from 1999 on learning-based testing, Peled, Vardi & Yannakakis [43, 44] combine three approaches for the verification of computer-based systems into what they call *black-box checking (BBC)*: *model checking*, which checks if a known state diagram model conforms to a specification, *conformance testing* (a.k.a. *model-based testing (MBT)*), which checks if a black-box system conforms with an abstract design, and *active automata learning* (a.k.a. *model learning*), which constructs a state diagram model of a

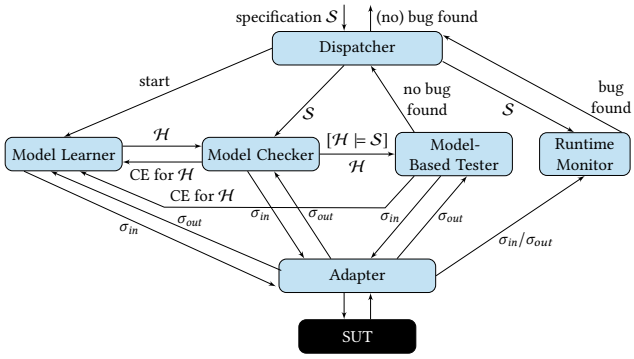


Figure 1: Black-box checking for specification $\mathcal{S}$ and system under test SUT.

black-box system by providing inputs and observing outputs. By combining these approaches, the authors obtain a method to check whether a black-box implementation satisfies a high-level specification. Conceptually, BBC is just a form of model-based testing. However, by learning the state-transition behavior of the SUT during testing, BBC is more effective than MBT algorithms in finding deep bugs in implementations for which only a high-level specification is available. In the rest of this paper, we will use the term black-box checking, but one could argue that the phrase *learning-based testing* coined by Meinke [38, 39] is more appropriate.

Figure 1 schematically shows how BBC works. First of all, it assumes an implementation (of a black-box system), referred to as the *System Under Test (SUT)*, that needs to be analyzed. It also assumes a (partial) specification $\mathcal{S}$, usually a conjunction of a number of requirements on the behavior of the SUT. An *Adapter* translates the abstract input and output symbols from the specification to concrete inputs and outputs that are accepted/produced by the SUT. The adapter makes BBC scalable to realistic applications, by using powerful abstractions [1, 55]. A *Dispatcher* orchestrates the activities of the different analysis tools used in the BBC approach. The dispatcher starts the analysis by activating the *Model Learner* [4, 30, 56], which builds a state-transition model $\mathcal{H}$ (an *hypothesis*) from results of *learning queries* (a.k.a. *output queries*): the learner sends sequences σ_{in} of inputs to the adapter/SUT, and in return receives sequences σ_{out} of outputs. The hypothesis $\mathcal{H}$ aims to describe how the SUT actually works. Whereas specification model $\mathcal{S}$ will often be small (e.g., a two state model describing that a *response* may only occur after a *request*), the hypothesis $\mathcal{H}$ will typically be much bigger, because the learning algorithm discovers — through systematic exploration — that states of the SUT reached via certain input sequences have different input-output behavior. Subsequently, BBC runs a *Model Checker* [6, 11] to check if hypothesis $\mathcal{H}$ *satisfies* specification $\mathcal{S}$, writen $\mathcal{H} \models \mathcal{S}$, meaning that all behaviors of $\mathcal{H}$ are allowed by $\mathcal{S}$. Two cases are considered:

(1) If $\mathcal{H} \models \mathcal{S}$ then a *Model-Based Tester* [7, 34, 52] will run *testing queries* to check whether $\mathcal{H}$ also is a correct model of the SUT: it sends sequences σ_{in} of inputs to the adapter/SUT, and checks if the resulting sequence σ_{out} of outputs agree with the prediction of $\mathcal{H}$. If a test fails, then the model learner uses this test to improve $\mathcal{H}$. If all tests pass then BBC terminates and reports that no evidence was found that the SUT violates specification $\mathcal{S}$.

(2) If $\mathcal{H} \not\models \mathcal{S}$ then there is a counterexample σ_{in} for which $\mathcal{H}$ produces an output not allowed by $\mathcal{S}$. BBC then performs an additional learning query σ_{in} on the adapter/SUT. If the resulting output σ_{out} agrees with the prediction by $\mathcal{H}$ then the SUT violates $\mathcal{S}$. Otherwise, $\mathcal{H}$ is not a correct model of the SUT, and the model learner uses the counterexample to further improve $\mathcal{H}$.

Finally, we included a *Runtime Monitor* [35] in Figure 1, as a minor extension of the BBC framework presented in [43, 44].[1] The runtime monitor checks, for all (learning and testing) queries performed on the SUT, whether the output σ_{out} produced in response to an input σ_{in} is allowed by specification $\mathcal{S}$. If an output violates $\mathcal{S}$ then a bug is reported. Use of a runtime monitor allows us to detect bugs faster: when a bug is encountered, it is reported immediately, rather than using it to further improve hypothesis model $\mathcal{H}$.

Surprisingly, in the large majority of applications of BBC, model checking is only used at the very end, when no counterexample can be found anymore for the latest hypothesis model (exceptions are, e.g., [37–39]). This contrasts with the original BBC proposal of [43, 44] which applies model checking for *all* hypotheses, also the intermediate ones. This is remarkable since, as observed by [43, 44], "intuitively it is clear that in many cases this method [only checking the final hypothesis] can be wasteful in that it does not take advantage of the property to avoid doing a complete identification". In a black-box setting we can never be sure about the correctness of learned models unless we are willing to make strong (typically unrealistic) assumptions, such as a bound on the number of states of the system. Therefore one might argue that, rather than learning models, the main goal of BBC is to find bugs in implementations as quickly as possible.

Some serious studies have been carried out to benchmark combinations of learning and testing algorithms for BBC [3, 26], but (surprisingly) there is no systematic evaluation of the ability of BBC to find bugs quickly. Only a few isolated results have been reported. Meinke & Sindhu [39], for instance, concluded for a single benchmark of an elevator system that the time required by BBC to discover a bug in the SUT is between 0.003% and 7% of the total time needed to completely learn the full model. The objective of this article is to provide a systematic evaluation of the effectiveness of BBC. In particular, we aim to answer the following questions:

RQ1 How effective is BBC (in terms of number of required queries) in detecting specification violations when compared with the effort required to learn the full model?
RQ2 How effective is BBC in detecting specification violations in situations where the full model cannot be learned?
RQ3 Does it help to add runtime monitoring to the BBC toolbox?
RQ4 How effective is BBC in detecting bugs when compared with traditional model-based testing algorithms.

[1] Monitors have also been added to BBC by Meijer & Van de Pol [37] but with a different purpose. They study BBC for general LTL formulas and propose to let the model checker consider the safety portion of an LTL property first and derive simpler counterexamples using monitors. So [37] uses monitors as part of the model checker, and not for runtime monitoring of learning and testing queries.

In our experiments we simulate the SUT from benchmark models that were obtained using automata learning in previous works.[2] From the Automata Wiki [41][3] repository, we selected 77 benchmark models with the following characteristics:

- All models were obtained via automata learning from real protocol implementations and real embedded controllers.
- All models are deterministic Mealy machines in which each input from a finite set I triggers a sequence of outputs taken from a finite set O.
- For all models, a specification was available consisting of a number of safety properties, specified either as a DFA over alphabet $I \cup O$, or as an LTL formula that we converted to a DFA over alphabet $I \times O^*$ using Spot [16].[4] The specifications that we consider are DFAs, but multiple outputs may be enabled in a single state, meaning that they allow for *observable nondeterminism* in the sense of [46, 57].

The rest of this article is structured as follows. Section 2 recalls the basic notations and definitions related to DFAs and Mealy machines that we use. Section 3 briefly introduces the case studies from which we obtained our benchmark models and specifications. Section 4 discusses our experimental setup, both for BBC and model-based testing. The results from our experiments are presented in Section 5. Finally, Section 6 presents our conclusions, and Section 7 discusses limitations of our approach and directions for future research.

2 Preliminaries

In this preliminary section, we first fix notation for partial maps and sequences, and then for DFAs and Mealy machines. Next, we describe how DFAs can be used to specify properties of Mealy machines, and a simple model checking approach for this setting. Finally, we briefly discuss our MBT approach.

2.1 Partial Maps and Sequences

We write $f : X \rightharpoonup Y$ to denote that f is a partial function from X to Y and write $f(x)\downarrow$ to mean that f is defined on x, that is, $\exists y \in Y : f(x) = y$, and conversely write $f(x)\uparrow$ if f is undefined for x. Function $f : X \rightharpoonup Y$ is *total* if $f(x)\downarrow$, for all $x \in X$. Often, we identify a partial function $f : X \rightharpoonup Y$ with the set $\{(x, y) \in X \times Y \mid f(x) = y\}$. The composition of partial maps $f : X \rightharpoonup Y$ and $g : Y \rightharpoonup Z$ is denoted by $g \circ f : X \rightharpoonup Z$, and we have $(g \circ f)(x)\downarrow$ iff $f(x)\downarrow$ and $g(f(x))\downarrow$. We use the Kleene equality on partial functions, which states that on a given argument either both functions are undefined, or both are defined with equal values for that argument.

An *alphabet* Σ is a finite set of symbols. A *word* over alphabet Σ is a finite sequence of symbols from Σ. We write ϵ to denote the empty word, and a to denote the word consisting of a symbol $a \in \Sigma$. If w and w' are words over Σ then we write $w\ w'$ to denote their *concatenation*. We write Σ^* to denote the set of all words over Σ, and Σ^+ to denote set $\Sigma^* \setminus \{\epsilon\}$. A word v is a *prefix* of a word w if there exists a word u such that $w = v\ u$. Similarly, v is a *suffix* of w if there exists a word u such that $w = u\ v$. A *language* over Σ is a subset of Σ^*. If L is a language over Σ, then we define $(L)^c$ as the language $\Sigma^* \setminus L$. A language L is *prefix closed* if $w \in L$ implies $v \in L$, for any prefix v of w.

2.2 DFAs and Mealy Machines

Definition 2.1 (DFA). A (partial) *deterministic finite automaton (DFA)* is a five-tuple $\mathcal{A} = (Q, \Sigma, \delta, q^0, F)$, where Q is a finite set of *states*, Σ is a set of *input symbols*, $q^0 \in Q$ is the *initial* state, $F \subseteq Q$ is a set of *final* states, and $\delta : Q \times \Sigma \rightharpoonup Q$ is a (partial) *transition function*. The transition function is inductively extended to words over Σ in the standard way, (for $w \in \Sigma^*$ and $a \in \Sigma$):

$$\begin{aligned} \delta(q, \epsilon) &= q \\ \delta(q, w\ a) &= \delta(\delta(q, w), a). \end{aligned}$$

A word $w \in \Sigma^*$ is *accepted* by $\mathcal{A}$ if $\delta(q^0, w) \in F$, and *rejected* by $\mathcal{A}$ if $\delta(q^0, w) \notin F$. The *language* accepted by $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all words accepted by $\mathcal{A}$. Two DFAs $\mathcal{A}$ and $\mathcal{A}'$ are *equivalent*, denoted $\mathcal{A} \approx \mathcal{A}'$, if they accept the same language. A DFA $\mathcal{A}$ is *prefix closed* if, for each state q and input a, $\delta(q, a) \in F$ implies $q \in F$. DFA $\mathcal{A}$ is *complete* if transition function δ is total.

It is easy to see that if $\mathcal{A}$ is prefix closed then $\mathcal{L}(\mathcal{A})$ is prefix closed. Each DFA can be turned into a complete DFA by adding a fresh (nonfinal) *sink state* and adding, for each missing transition, a transition to that sink state.

Definition 2.2 (Completion of a DFA). Let $\mathcal{A} = (Q, \Sigma, \delta, q^0, F)$ be a DFA. Then Complete($\mathcal{A}$) is the complete DFA $(Q', \Sigma, \delta', q^0, F)$, where $Q' = Q \cup \{q_s\}$ and, for all $q \in Q'$ and $a \in \Sigma$,

$$\delta'(q, a) = \begin{cases} \delta(q, a) & \text{if } \delta(q, a)\downarrow \\ q_s & \text{otherwise.} \end{cases}$$

It is straightforward to verify that $\mathcal{A} \approx$ Complete($\mathcal{A}$).

Whereas the output of a DFA is limited to a binary signal (accept/reject), a Mealy machine associates a more general output to each transition. The definition below, adapted from [21], associates a sequence of outputs from some alphabet to each transition. Such machines are quite useful to model the behavior of protocol entities.

Definition 2.3 (Mealy machine). A (partial) *Mealy machine* is a tuple $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$, where I and O are alphabets of *input* and *output symbols*, respectively, Q is a set of *states* containing the *initial* state q_0, $\delta : Q \times I \rightharpoonup Q$ is a (partial) *transition function*, and $\lambda : Q \times I \rightharpoonup O^*$ is a (partial) *output function*. We require, for all $q \in Q$ and $i \in I$, that $\delta(q, i)\downarrow$ iff $\lambda(q, i)\downarrow$. A Mealy machine is *complete* if δ (and hence λ) is total.

[2] The main advantage of simulating the SUT from benchmark models, rather than using real implementations is that it reduces the time required for experiments. For the SSH protocol, it took two days to learn a model from a real implementation [21]. About half the time was used for conformance testing the final hypothesis. Based on this, we believe that simulating the learned SSH model is a good proxy for the real implementation (and likewise for the other protocol models). Code for the controllers from the RERS benchmarks was generated from deterministic FSMs [62], and this enabled proof that the learned models are correct. Learning an SSH model in a simulated setting only takes about 40 seconds, so about 2000 times as quick as learning from the real thing (ignoring testing the final model). Running all experiments from this paper took us about six days; we therefore estimate that running our experiments on real implementations would take decades.

[3] https://automata.cs.ru.nl/

[4] In the original paper of Peled, Vardi & Yannakakis [43, 44], Büchi automata are used as specifications and liveness properties are considered. However, in order to establish soundness of their BBC procedure, they need to assume a known bound on the number of states of the SUT. As pointed out by Meijer & Van de Pol [37] this can be either dangerous (if the guessed bound is too low) or inefficient (if the bound is too high). Meijer & Van de Pol [37] propose an alternative BBC procedure for liveness properties that requires the ability to test for equality of SUT states, but this is not truly black-box. We therefore decided to focus on safety properties in our study.

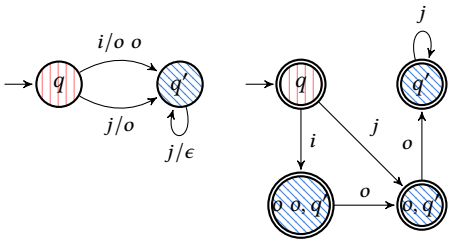


Figure 2: A Mealy machine (left) and the corresponding DFA (right).

2.3 Model Checking

As part of BBC, the model checker verifies if a hypothesis $\mathcal{H}$ satisfies specification $\mathcal{S}$. Our hypotheses are Mealy machines, while we use Deterministic Finite Automata (DFA) for specifications. Inspired by [21], we translate our Mealy machines into DFAs, such that satisfaction of $\mathcal{H}$ to $\mathcal{S}$ can be checked using standards available for DFAs [29].

To translate from Mealy machines to DFAs, the idea is to insert auxiliary states (w, q) in between the source q' and target q of a transition, in which first the outputs from w are performed before jumping to state q.

Definition 2.4 (Mealy machine to DFA). Let $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ be a Mealy machine, with $I \cap O = \emptyset$. Then MealyToDFA($\mathcal{M}$) is the partial DFA $(Q', I \cup O, \delta', q^0, Q')$ where, for all $q \in Q$, $i \in I$, $o, o' \in O$ and $w \in O^*$: $\delta'(q, o)\uparrow$, $\delta'((o\ w, q), i)\uparrow$ and

$$\begin{aligned} Q' &= Q \cup Q_{aux} \\ Q_{aux} &= \{(v, q) \in O^+ \times Q \mid \exists q' \in Q, j \in I : \\ &\qquad \delta(q', j) = q \text{ and} \\ &\qquad v \text{ suffix of } \lambda(q', j)\} \end{aligned}$$

$$\delta'(q, i) = \begin{cases} \delta(q, i) & \text{if } \lambda(q, i) = \epsilon \\ (\lambda(q, i), \delta(q, i)) & \lambda(q, i) \in O^+ \\ \text{undefined} & \lambda(q, i)\uparrow \end{cases}$$

$$\delta'((o\ w, q), o') = \begin{cases} \text{undefined} & \text{if } o \neq o' \\ q & \text{if } o = o' \text{ and } w = \epsilon \\ (w, q) & \text{otherwise.} \end{cases}$$

Figure 2 gives an example of our translation from Mealy machines to DFAs. Note that MealyToDFA($\mathcal{M}$) is a labeled transition system with inputs and outputs in the sense of Tretmans [52], but of a restricted form. States of the DFA either have a *single* outgoing output transition, or zero or more outgoing input transitions. Moreover, at most finitely many consecutive output transitions are possible from any state. It is easy to see that, given I and O, we may fully retrieve $\mathcal{M}$ from MealyToDFA($\mathcal{M}$). Our translation improves on the one presented in [21], which turns the Mealy machine of Figure 2 into a DFA with 5 states.

Definition 2.5 (Specification). Consider a Mealy machine $\mathcal{M}$ with inputs I and outputs O. A *specification* for $\mathcal{M}$ is a prefix closed DFA $\mathcal{S}$ over alphabet $I \cup O$. We say that $\mathcal{M}$ *satisfies* specification $\mathcal{S}$ if $\mathcal{L}(\text{MealyToDFA}(\mathcal{M})) \subseteq \mathcal{L}(\mathcal{S})$.

The specifications for our benchmark models need to be massaged a bit to turn them into the above format. For the BLE and RERS case studies the specifications are LTL formulas. Using the Spot tool [16], we translate these to equivalent DFAs over alphabet $I \times O$. By splitting each (i, o)-transition into an i-transition followed by an o-transition, we can translate these DFA into (prefix-closed) DFAs over alphabet $I \cup O$. For the SSH and DTLS case studies, the specifications are given in terms of "bug automata". These are DFAs over alphabet $I \cup O$, but with the roles of final/nonfinal states interchanged: accepting runs correspond to undesired behavior of the SUT. By complementing these bug automata, we obtain specifications in the sense of our Definition 2.5.

A model checker can efficiently check that $\mathcal{M}$ satisfies $\mathcal{S}$, using the fact that for all DFAs $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ iff $\mathcal{L}(\mathcal{A}) \cap (\mathcal{L}(\mathcal{B}))^c = \emptyset$. Both intersection and complement can be easily defined for DFAs:

Definition 2.6 (Complement). Let $\mathcal{A} = (Q, \Sigma, \delta, q^0, F)$ be a complete DFA. Then the *complement* of $\mathcal{A}$, denoted $(\mathcal{A})^c$, is the DFA $\mathcal{A} = (Q, \Sigma, \delta, q^0, Q \setminus F)$.

Definition 2.7 (Product). Let $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, q_1^0, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, q_2^0, F_2)$ be two DFAs. Then the *product* of $\mathcal{A}_1$ and $\mathcal{A}_2$, denoted $\mathcal{A}_1 \parallel \mathcal{A}_2$, is the DFA $(Q_1 \times Q_2, \Sigma, \delta, (q_1^0, q_2^0), F_1 \times F_2)$, where for all $q_1 \in Q_1$, $q_2 \in Q_2$ and $a \in \Sigma$,

$$\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a)).$$

It is well-known that for complete $\mathcal{A}$, $\mathcal{L}((\mathcal{A})^c) = (\mathcal{L}(\mathcal{A}))^c$ and, for DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ with the same alphabet, $\mathcal{L}(\mathcal{A}_1 \parallel \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$, see, e.g., [29]. Thus $\mathcal{M}$ satisfies $\mathcal{S}$ iff

$$\mathcal{L}(\text{MealyToDFA}(\mathcal{M}) \parallel (\text{Complete}(\mathcal{S}))^c) = \emptyset.$$

Emptiness of the language accepted by a DFA can be decided in time linear in the size of the DFA [29].

2.4 Model-Based Testing

In this paper, MBT is used in two different ways. First, MBT is used as part of the BBC procedure, namely to check conformance of the current hypothesis to the SUT. Secondly, we use MBT as a standalone method to find bugs in the SUT. This way, we can compare the bug finding capabilities of BBC with standalone MBT.

In MBT, tests are derived from a model. In MBT as part of BBC the model is the hypothesis. Standalone MBT uses the specification as its model: it uses the prefix-closed DFAs over the alphabet $I \cup O$, where a transition is either labeled with an input or an output label. Therefore, these DFAs can be trivially translated to a Labeled Transition Systems, for which standard test derivation is defined as usual [52], i.e., a test either: (1) supplies an input specified by the model, (2) waits to observe an output – if the received output is not the output specified by the model, the test stops with verdict fail, or (3) stops with verdict pass, when the stop condition, e.g., number of test steps, has been satisfied.

While testing one keeps track of the current state of the model according to the supplied inputs and observed outputs. For selecting the next input of a test, existing MBT test generation strategies can be used, e.g., to select an input randomly, or to select a different input than already tried. Note that, when MBT is used as standalone method, a test fails whenever the SUT generates an output that is

not allowed by the specification. Thus, essentially, standalone MBT already uses runtime monitoring to implement its test oracle, and adding runtime monitoring would not change the results.

3 Case Studies

We selected case studies from several published papers for which both models of SUTs were available, as well as properties that could be converted to DFAs. The SUT models were available on the Automata Wiki [41]; specifications were obtained from the paper and corresponding artefacts. The case studies vary in size and application area, so that we have a good basis to draw conclusions.

BLE. In [48], Pferscher et al. use an active automata learning approach to detect anomalies and security vulnerabilities in five distinct System on Chip (SoC) Bluetooth Low Energy (BLE) devices, all of which implement the BLE 5 standard. Pferscher et al. found several crashes, anomalies, and a security vulnerability with their fuzzing-enhanced learning process.

SSH (LTL). In [22], Fiterau-Brostean et al. use active automata learning and model checking to verify several security and functional requirements against three SSH server implementations (Bitvise 7.23, Dropbear v2014.65 and OpenSSH 6.9p1-2). They use the NuSMV [10] model checker to automatically check their LTL formalizations of 12 requirements imposed by RFCs that describe the second version of the SSH protocol (i.e., SSHv2) against the final Mealy machine models they learned for their selected implementations. They found that each of the implementations under test violated at least one of their selected functional requirements.

SSH (DFA). In [21], Fiterau-Brostean et al. describe expected bugs and vulnerabilities as DFAs, and check these during automata learning. They apply their method to three more recent versions of the SSH server implementations (i.e., BitVise 8.49, DropBear v2020.81 and OpenSSH 8.8p1).

DTLS (Client & Server). Fiterau-Brostean et al., also apply their method from [21] to 16 DTLS client implementations and 17 DTLS server implementations (there were multiple versions of the same DTLS implementations).

RERS. The Rigorous Examination of Reactive Systems (RERS)-challenges are semi-annual open competitions in which participants attempt to solve a given set of reachability and verification problems. The goal behind these competitions is to encourage the use of new combinations of various tools and approaches and to provide a basis of comparison for the approaches used by the participants. The 2019-edition of the RERS challenge [31] contained an industrial track with benchmark programs that are based on Mealy machine models of controller software provided by the company ASML. We used all thirty SUTs of the RERS 2019 industrial LTL challenge, along with their accompanying LTL specifications.[5]

4 Experimental Setup

We first explain our experimental setup to evaluate BBC, and its variant in which only the final hypothesis is model-checked. Then we explain our experimental setup for (traditional) MBT.

4.1 Black-Box Checking and Model Learning

In this work, we determine the efficacy of BBC by comparing its efficiency to that of "basic" model learning. The main difference between the basic variant and BBC is that the former does not involve the use of a model checker[4, 30, 43, 44, 56], except possibly at the very end when learning has finished. In our experiments, basic model learning finishes once the Model-Based Tester concludes that the current hypothesis $\mathcal{H}$ is equivalent to the SUT. The basic model learning setup also does not involve the use of a Runtime monitor.

Our BBC and model learning approaches both offer support for the use of multiple properties at once. Let $\mathcal{S} = (P_1, \ldots, P_n)$ be a tuple of $n \in \mathbb{N}$ properties. When BBC or model learning uses the Model Checker, it checks every hypothesis $\mathcal{H}$ that it receives against each property in $\mathcal{S}$ in the order of inclusion. For each counterexample σ^i_{in} for $P_i \in \mathcal{S}$ that it finds, the Model Checker compares the output $\sigma^{i,\mathcal{H}}_{out}$ of $\mathcal{H}$ against the output $\sigma^{i,SUT}_{out}$ of the SUT. If $\sigma^{i,\mathcal{H}}_{out} = \sigma^{i,SUT}_{out}$, then the Model Checker passes σ^i_{in} to the Dispatcher, along with the associated property P_i, and P_i is removed from $\mathcal{S}$. If $\sigma^{i,\mathcal{H}}_{out} \neq \sigma^{i,SUT}_{out}$, then σ^i_{in} is called a spurious counterexample for P_i, and the Model Learner will improve $\mathcal{H}$ for σ^i_{in} once the Model Checker has checked $\mathcal{H}$ against all $\{P_j \in \mathcal{S} \mid i < j \leq n\}$. The Model-Based Tester is used iff the model checker didn't find any counterexamples, spurious or otherwise. The Runtime Monitor checks the SUT against precisely the properties in $\mathcal{S}$. It reports any $P_i \in \mathcal{S}$ for which it finds a SUT counterexample to the Dispatcher, after which P_i is removed from $\mathcal{S}$.

Our approach to using multiple properties has the consequence that all properties may affect one another: any spurious Model Checker counterexample is used to refine the hypothesis. The hypothesis then changes, which affects the way in which BBC will look for counterexamples for the properties (still) in $\mathcal{S}$. The hypothesis refinements that BBC performs for spurious counterexamples can therefore affect the time that it takes to complete the BBC process, which can thus deviate from the time that it would take to complete model learning followed by model checking for the same SUT and properties. Both approaches report upon their completion that they found no evidence that the SUT violates any of the properties that are still in $\mathcal{S}$. Properties that hold for the SUT will always remain in $\mathcal{S}$ when BBC or model learning finishes. The two approaches may therefore take a different amount of time to report for the properties satisfied by the SUT that they didn't find evidence that the SUT violates them.

We implemented the BBC and basic model learning approaches in a single codebase written in Python 3.11.14 on top of version 1.5.1 of the AALpy [40] library for active automata learning. We used version 2.13 of the LTL and ω-automata manipulation and model checking library Spot [16] to convert the LTL properties into monitors. For performing experiments, we used Singularity[6].

We used the active automata learning algorithm $L^\#$ because its performance is competitive with that of several alternatives [53]. We use separating sequences for $L^\#$'s extension rule because this is the default in AALpy. For most experiments, we use Adaptive Distinguishing Sequences (ADS) for the separation rule as this is AALpy's

[5] https://rers-challenge.org/2019/index.php?page=industrialProblemsLTL#

[6] https://sylabs.io/docs/

default setting. The exception is RERS, where we used separating sequences because we found that using ADS with experiments that were otherwise unchanged made the hypothesis refinements and thereby the BBC process in the RERS models considerably slower.

We fixed the model-based tester for BBC to Hybrid-ADS [51] because it is a recent, state-of-the-art approach that is available as a tool. We based our configuration for Hybrid-ADS on [53]. As such, we set the operation mode to "random", the prefix mode to "buggy", the number of extra states to check for (minus 1) to 10, and the expected number of random infix symbols to 10. We repeat each of our experiments for 50 random seeds to account for the randomness introduced by our use of Hybrid-ADS. We measure the time that each seed takes with the `perf_counter_ns`-function from the `time`-module of Python's standard library.

Hybrid-ADS produces an unbounded number of testing queries when used in random mode. Performing another correctness check when the current hypothesis is equivalent to the SUT would take an infinite amount of time. In [53], the authors skip this final correctness check if they determine that the current hypothesis is already equivalent to the SUT. We follow this approach because the number of testing queries that one would perform to look for a difference between a correct hypothesis and the SUT is always both finite and ultimately arbitrary.

We followed the approach of Kruger et al. [32], by using a step budget to further reduce the time taken by certain BBC experiments. If an experiment is about to exceed this budget, BBC first terminates its current operation, and then uses the model checker to look for counterexamples for the properties for which it hadn't found one. This use of the model checker does imply that the step budget can be exceeded, since the model checker will still verify any counterexamples that it finds against the SUT. Whenever we used step budgets for an SUT, we did so because we found that it should take us considerably more than a week to obtain all of our results for it. We then used a range of step budgets that spanned from 10^3 to 10^8 and selected the results for the largest budget for which we obtained results in a timely fashion.

To evaluate our approach, we run our experiments on SUTs simulated from Mealy machines (stored as DOT files) obtained with automata learning in previous work (see Section 3). This way our experiments are not hindered by, e.g., response times of a real SUT. For each case study, multiple SUT models were available, as they were obtained from different real implementations. All properties available for the case studies were converted from their format, which ranged from natural language to LTL formulas, to DFAs. We performed experiments for all combinations of SUTs and their associated properties.

We made the following decisions in obtaining the case studies:

BLE. We used all available SUTs and properties which we could convert into LTL properties.

SSH (LTL). We use 9 of the 11 LTL properties for our evaluation. Property 3, Property 4 and Property 9 are excluded, because they each use either NuSMV's *O* (once), or NuSMV's *S* (since) LTL operator, neither of which are supported by Spot[7]. Property 8 relies on the ability to check whether the SUT is in its initial state, a requirement that cannot be satisfied in our black-box setting. We therefore replaced this property with an LTL property that we based on a DFA from the SSH (DFA) case study. Fiterau et al. designed that DFA to capture the same requirement of the SSHv2 specification as Property 8. It does so in a way that doesn't rely on explicit knowledge of the SUT's current state, which allows it to work in our black-box setting.

SSH (DFA) & DTLS (Client & Server). We obtained the SUTs and associated properties used in Fiterau et al.'s [21] from their archived software artifact on [23]. We only used the SUTs for which the artifact specified which of their properties hold and which do not. These were precisely the SUTs covered in [21].

RERS. We used the 30 provided models, and the safety properties among their accompanying sets of 20 LTL properties. The number of safety properties, and thereby of the properties that we used per model ranges from 9 to 18, with a mean of 13.7 and a standard deviation of 1.9.

Table 1: The total number of SUT models, mean and standard deviation of the number of states of the SUT models, the total number of properties that violate the SUTs, and the mean and standard deviation of the number of properties violated by the individual SUTs, per case study.

Case study	# SUTs	# SUT states		# SUT violations		
		Mean	Stdev	Total	Mean	Stdev
BLE	8	7.6	4.5	1	0.1	0.4
SSH (LTL)	3	42	20.8	6	2	1.0
SSH (DFA)	3	33.7	11.4	11	3.7	1.5
DTLS Client	16	84.6	122.8	66	4.1	3.4
DTLS Server	17	90.7	238.0	44	2.6	3.1
RERS	30	432.7	835.6	182	6.1	2.4

Table 1 shows some general information about the case studies. For each case study, it shows the number of SUTs, the means and standard deviations of the number of SUT states and the total number of properties that are violated by the SUTs, followed by the means and standard deviations of the number of violated properties for each individual SUT. BBC is used to determine whether a black-box SUT satisfies a given set of properties. The user wouldn't know beforehand which of the properties are satisfied by the SUT, and which are not. We therefore included the properties that are satisfied by the SUT in our experimentation. The inclusion of these properties can affect the results, since any model checking counterexamples that are found for these properties will trigger a hypothesis refinement.

To make for a fair comparison, we use the same settings for the model learning experiments as for their corresponding BBC experiments. The only difference between them is in the way that the Model Learner, Model-Based Tester, Model Checker and Runtime Monitor are (or are not) used to follow either the BBC schematic of Figure 1 or the model learning variant.

[7] https://spot.lre.epita.fr/concepts.html#ltl

We performed the black-box checking and model learning experiments on a computer with an AMD Epyc 7642 processor. Every experiment was constrained to 2 processing cores and 4Gb of RAM.

4.2 Model-Based Testing

In order to evaluate BBC's performance against a baseline, we applied (standalone) model-based testing on a subset of the evaluated benchmarks. We selected a subset of experiments focusing on bugs that require only low to moderate effort for BBC to produce counterexamples. This restriction is motivated by preliminary results indicating significantly worse performance of MBT.

For the MBT implementation, we use Lattest[8] , a Haskell library for MBT. As a test selection strategy, we implemented a tester that combines a random walk with the aim of increasing transition coverage. Specifically, we add memory to store the input sequences that were tested already by some test of the test suite. At each step of a test, a new input is chosen randomly from the set of inputs that has not been tried yet after this sequence. The test stops either when it fails or when it reaches the configured maximum number of steps.

We chose our random walk strategy with memory for transition coverage as an optimum in between a simple random walk and more sophisticated techniques, such as n-complete test suites [15, 34, 58]. Given that a test from an n-complete test suite always starts with an access sequences to a specification state, we noted, also from initial experiments where we tried tests of the n-complete test suite, that the access sequences of the specification where not matching with access sequences to a respective state in an implementation. The reason for this is that the specification consists of properties that typically do not consider a trace from the initial state, but only the few relevant inputs and outputs considered by that property. For example, it was expecting to see a response after a certain input, but that response was only observed if before the input some other actions were done. Consequently, a direct access sequence which just provides the respective input would not work to reach the state where the response transition is enabled. Summarizing, the high-level and partial nature of the specification omits details that sophisticated algorithms need to efficiently explore the implementation and thus efficiently find bugs.

Initial experiments also confirmed that random walk could find few bugs compared to BBC. With our memory test strategy, by remembering sequences, we have a somewhat course representation of traces of the implementation (still much courser than hypothesis models), so that MBT is able to somewhat compete with BBC, while still representing a straightforward MBT test strategy.

As done in BBC, multiple properties can be tested on the same SUT simultaneously. Given a set of properties, the tester constructs a conjunction of the corresponding specifications. When a bug in the SUT is detected, the violated property is identified, and the specification is regenerated excluding that property, to continue testing. Shadowing of deeper bugs by shallower ones is thus avoided.

The subset of experiments executed for MBT consists of 48 combinations of SUTs and properties, therefore, 48 potential bugs to be found. Each experiment is repeated 50 times, with different seeds, to account for the randomness introduced by the tester. For each experiment, the number of test cases for the test suite is 10 times the number of queries it takes for the BBC implementation to find the corresponding bug. The number of test steps within each test is defined as twice the number of states of the specification, to aid the identification of deeper bugs. We also considered an alternative bound for the stopping criteria of a test, by looking at the length of traces for BBC to find bugs, but since we then would use white-box knowledge, we decided that a generic bound based on the specification would be fairer since that is truly black-box. Afterwards, we checked that this practical bound exceeded the number of steps that BBC needed to find bugs.

The specifications for these experiments were constructed from LTL formulas as described in Definition 2.5, or from the DFAs, depending on the benchmark. As in BBC, the implementations described in the DOT files are simulated by a Python script. Both the tester and the Python-based SUT are executed inside Docker[9] containers. MBT experiments were executed on a computer with an Intel i7 processor. Each instance of the tester was constrained to 1 processing core and 2Gb of RAM.

5 Results

In this section, we present the results of our experiments, addressing each of the questions posed in Section 1. We consider 77 SUT models, for a total of 310 pairs of models with accompanying violated properties. Our use of step budgets kept us from finding all 310 bugs. In total, our BBC approach found 304 (98%) of the bugs across all 50 random seeds. The RERS case study was the only case study for which not all bugs were found in all seeds: BBC found a mean of 96% of the bugs across the 50 seeds, with a standard deviation of 10%. Our BBC found a mean of 278 (90%) of the 310 bugs per seed, with a standard deviation of 5 bugs.

5.1 BBC vs. Model Learning

Of the 65 models across which BBC found at least one bug, 32 (49%) were fully learned by model learning. This accounts for 118/310 (38%) of the bugs. Figure 3 shows for each of these 118 properties the mean number of queries that BBC and model learning required to find the bugs across all 50 seeds. The figure shows that for all bugs, the mean number of queries required by BBC was lower than that required by model learning. Model learning required a mean of 251,733 queries for these 118 properties, compared to a mean of 4,047 (2%) queries for BBC. For the 118 properties, BBC took a mean of 3% of the queries required by model learning, with a standard deviation of 7%. Table 2 shows the minimum, maximum, mean and standard deviation of the percentages for each case study. It shows that across all SUTs and properties, BBC needed at most 53% of the queries required by model learning.

Note how in Figure 3, some data points for bugs from the same case studies appear in vertical lines. These lines are formed by properties that belong to the same model; with model learning, all bugs for the same model take an almost equal number of queries to find, since they are only found once the models are fully learned[10].

[8] https://github.com/ramon-janssen/lattest

[9] https://www.docker.com/

[10] We say that they take an *almost* equal number of queries to find, because the Model Checker checks every counterexample found after model learning against the SUT with an additional learning query, as we explain in Section 1.

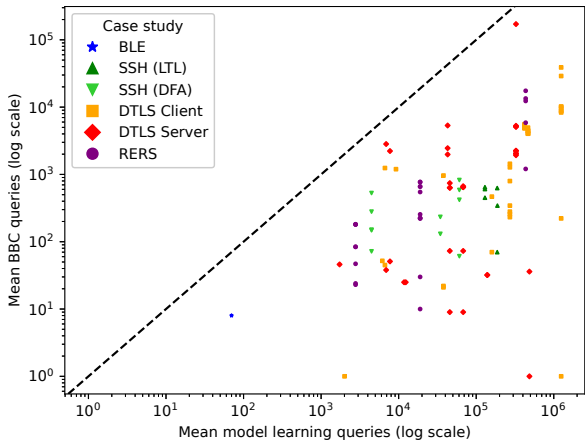


Figure 3: Scatter plot showing the number of queries needed for BBC to find a bug, relative to the number of queries needed to find the bug with model learning, in log scale.

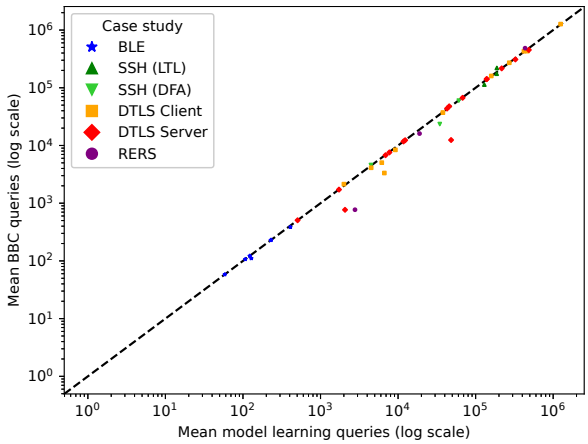


Figure 4: Scatter plot showing the mean number of queries needed for BBC to finish for the SUTs that satisfy at least one property, compared to model learning, in log scale.

Table 2: The minimum, maximum, mean and standard deviation of the percentages of the number of queries BBC took when compared to model learning for the bugs from Figure 3, per case study.

Case study	Min	Max	Mean	Stdev
BLE	12%	12%	12%	—
SSH (LTL)	0%	0%	0%	0%
SSH (DFA)	0%	12%	3%	4%
DTLS Client	0%	19%	2%	3%
DTLS Server	0%	53%	5%	12%
RERS	0%	7%	3%	2%

Table 3: The minimum, maximum, mean and standard deviation of the percentages of the number of queries BBC took to finish when compared to model learning for the SUTs of Figure 4, per case study.

Case study	Min	Max	Mean	Stdev
BLE	87%	100%	97%	5%
SSH (LTL)	88%	117%	100%	16%
SSH (DFA)	68%	103%	90%	19%
DTLS Client	50%	107%	92%	16%
DTLS Server	26%	104%	91%	23%
RERS	28%	112%	75%	43%

We explain in Section 4 why it may take BBC and model learning a different amount of time to finish and, therefore, a different amount of time to report for the properties which hold for the SUT that they didn't find any evidence to the contrary. There were 43 SUTs for which we fully learned the model with model learning, and for which the SUT satisfies at least one of the properties. Figure 4 shows for each of these 43 SUTs the mean number of queries that BBC and model learning required to finish across all 50 seeds. Model learning required a mean of 121,945 queries to fully learn these models, compared to a mean of 120,833 (99%) queries for BBC. For these 43 SUTs, BBC took a mean of 92% of the queries required by model learning, with a standard deviation of 20%. Table 3 shows the minimum, maximum, mean and standard deviation of the percentages for each case study. It shows that the mean number of queries that were needed to finish was lower for BBC than for model learning, for all SUTs across all six case studies. It also shows that all but one of the case studies have at least one SUT for which BBC took more queries to finish than model learning. In the most extreme case, BBC took 117% of the queries used by model learning.

Note how Figure 4 contains six data points for the BLE case study, while Figure 3 has only one. This is because our only BLE property holds for six of the seven BLE SUTs. The six SUTs for which it holds show up in Figure 4, while the SUT for which it doesn't hold is only represented in Figure 3. We noted for Figure 3 that vertical lines of data points for the same case study are a consequence of these points corresponding to different properties for the same SUTs. These lines don't show up in Figure 4 because the data points in Figure 4 each represent a distinct SUT.

We performed an ablation study to determine the effect of our runtime monitoring on BBC's performance. To this end, we repeated each BBC experiment with runtime monitoring disabled. Unmonitored BBC found the same 304 bugs found by (monitored) BBC. Figure 5 shows the mean number of queries that BBC and unmonitored BBC needed to find these bugs across the 50 seeds. This took unmonitored BBC a mean of 279,200 queries, while BBC required a mean of 274,860 (98%) queries. For the 304 properties, BBC took a mean of 79% of the queries required by unmonitored BBC, with a standard deviation of 30%. Table 4 shows the minimum, maximum, mean and standard deviation of the percentages for each case study. It shows that it is possible for BBC to need more queries to find a given bug than unmonitored BBC. This is a consequence of the way in which our BBC approach handles multiple properties at once. We explain in Section 4 that BBC stops model checking and runtime monitoring against any properties for which the Runtime Monitor finds counterexamples. The Model Checker can then no longer find spurious counterexamples for these properties. This

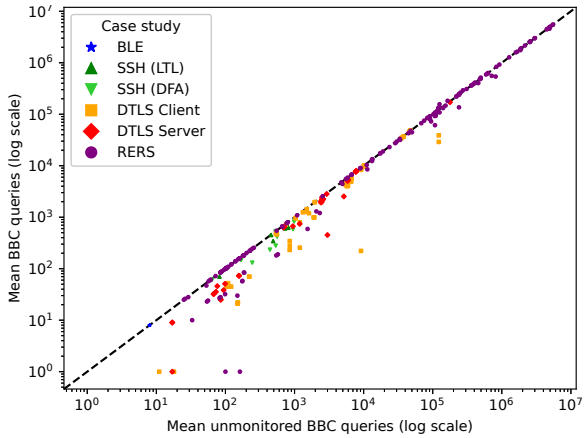


Figure 5: Scatter plot showing the number of queries needed for BBC to find a bug with runtime monitoring enabled, relative to the number of queries BBC required with runtime monitoring disabled, in log scale.

Table 4: The minimum, maximum, mean and standard deviation of the percentages of the number of queries BBC took when compared to unmonitored BBC for the bugs of Figure 5, per case study.

Case study	Min	Max	Mean	Stdev
BLE	100%	100%	100%	—
SSH (LTL)	70%	99%	82%	10%
SSH (DFA)	46%	99%	73%	20%
DTLS Client	2%	101%	69%	31%
DTLS Server	0%	100%	71%	28%
RERS	0%	125%	85%	31%

prevents the hypothesis refinements that would otherwise be performed for these counterexamples from taking place, which can affect the number of queries that are required to find the remaining bugs.

Figure 6 shows for each case study the number of states the hypotheses had when BBC found the bugs. The colored markers indicate the mean number of states across all 50 seeds; the colored lines that cross the markers show the standard deviations. The gray bars indicate the mean number of states of the SUT models, which have a mean of 210 states across the six case studies. BBC needed to learn a mean of 17 (8%) states to find the bugs, with a standard deviation of 19 states.

The processing overhead incurred by BBC is minor. Generating DFAs from LTL formulas, which we only need to do once, takes less than 4 seconds for our benchmarks. The key overhead of BBC consists of model checking the intermediate hypotheses. Across our experiments, model checking accounted for a mean of 9% of the execution time. The largest benchmark that we considered has only 3,966 states, which is really small from a model checking perspective. Note that hypotheses can never be larger than the SUT model.

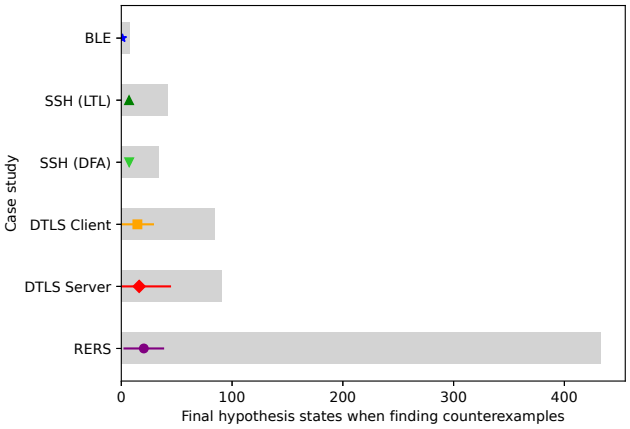


Figure 6: Bar plot comparing the mean and standard deviation of the number of states BBC needed to learn to find the bugs of the six case studies, along with the mean of the number of states in their SUTs.

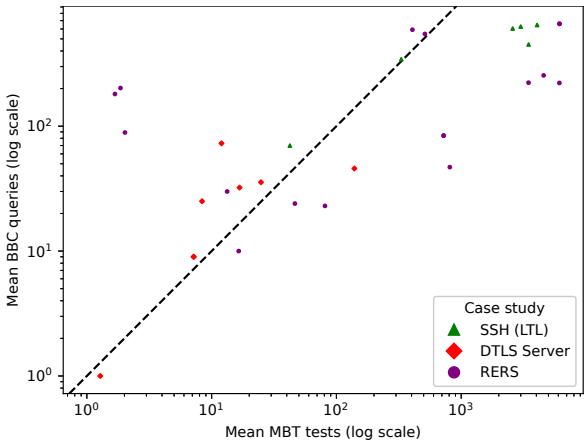


Figure 7: Scatter plot showing the mean number of queries needed for BBC to find a bug, compared to the mean number of tests MBT required to find the same bugs, in log scale.

5.2 BBC vs. MBT

Finally, we compare BBC's performance for bug finding with standalone MBT, for a subset of the benchmarks. Exploratory experiments showed that MBT consistently needed way more steps (inputs) to find counterexamples. In our experiments, we gave MBT 10 times the step-budget of BBC. We selected a subset of benchmarks for which BBC found a counterexample within a reasonable number (< 1 million) of steps. Across all 50 seeds, MBT was able to find 32 (67%) of the 48 bugs, while BBC found all of them.

Figure 7 shows the mean numbers of tests that MBT required, and the mean numbers of queries that BBC required to find these 32 bugs. MBT used a mean of 1,358 tests to find them, compared to a mean of 216 (16%) queries for BBC. For the 32 bugs, BBC took a mean of 896% of the queries, compared to the number of tests required by MBT, with a standard deviation of 2, 715%.

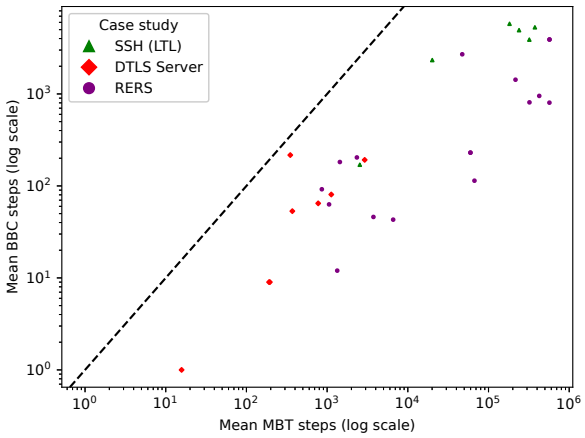


Figure 8: Scatter plot showing the mean number of steps needed for BBC to find a bug, compared to the mean number of steps MBT required to find the same bugs, in log scale.

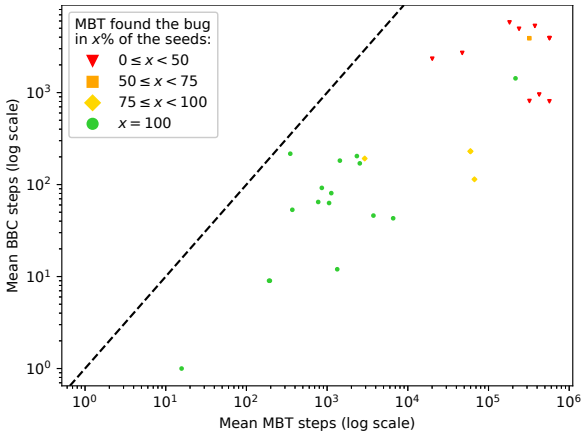


Figure 9: Scatter plot showing the mean number of steps needed for BBC to find a bug, compared to the mean number of steps MBT required to find the same bugs, in log scale.

We also compared the efficiency of the two testing methods on the number of steps (i.e., input events) used to find the bugs. Figure 8 shows the mean number of steps that MBT and BBC required to find the 32 bugs found by MBT, across all 50 seeds. The figure shows that for all of the 32 bugs, the mean number of steps required by BBC was lower than that required by MBT. MBT required a mean of 127,137 steps to find these bugs, compared to a mean of 1,212 (1%) for BBC. For the 32 bugs, BBC took a mean of 6% of the steps required by MBT, with a standard deviation of 11%.

BBC not only found more bugs quicker, but also did so consistently. MBT only reported 17 (53%) of the 32 bugs that it found consistently across all 50 seeds, whereas BBC managed to find all 48 bugs for all of its 50 seeds. Figure 9 presents again the comparison of number of steps required for both BBC and MBT to find each bug, but this time, the coloring of each point represents how consistent MBT was in finding the bug. The results indicate that MBT consistently finds shallow bugs (i.e., those that take few test steps), while deep bugs are only occasionally reported.

These results show that BBC consistently outperforms MBT in both speed and reliability. Even when MBT approaches BBC's performance in certain benchmarks, it lacks consistency. This variability is originated in our implementation's strong dependence on randomness, i.e., the initial seed, even when augmented with memory. As a result, MBT may occasionally discover faults quickly, but such outcomes are not consistent or reproducible.

6 Conclusions

BBC is quicker than learning the full model before model checking (RQ1). Our systematic evaluation confirmed the intuition of Peled, Vardi & Yannakakis [43, 44] that bugs are detected (much) quicker when, as part of the BBC loop, model checking is applied for all intermediate hypotheses, rather than just for the full model. Monitored BBC generally took fewer queries to finish than model learning, though this wasn't the case for all SUTs.

BBC finds bugs when the full model cannot be learned (RQ2). Even when the full model is too large and cannot be learned, BBC is still able to detect many violations of the specification. In particular, using BBC, we managed to detect 96% of the safety properties violations in the challenging RERS 2019 industrial LTL benchmarks. This improves (what we believe to be) the best previous result for this benchmark collection by Kruger, Junges & Rot [32], who succeeded to learn partial models for 23 benchmarks (they did not consider the 7 largest benchmarks), and full models for 15 benchmarks. Here we should emphasize that the goal of [32] was to learn the full models rather than finding specification violations.

Adding runtime monitoring helps (RQ3). We also found that adding runtime monitoring to the BBC toolbox generally leads to a decrease of the number of queries required to detect bugs.

BBC is more effective than MBT for high-level specifications (RQ4). Our results show that BBC is more effective than existing MBT algorithms in finding (deep) bugs in implementations when only a high-level model is available. MBT [7, 15, 34, 52] has been extensively researched and a reasonable number of commercial and open-source tools exist. MBT has proven to be quite effective when test are generated from "low level" formal models whose behavior is close to the behavior of the SUT, in the sense that all states of the SUT can be reached by first doing an access sequence for a state in the formal model, followed by a few randomly selected inputs. In fact, Vaandrager & Melse [54] observe that prominent MBT approaches, such as the W-method of Vasilevskii [59] and Chow [9], the Wp-method of Fujiwara et al. [25], and the HSI-method of Luo et al. [36] and Petrenko et al. [47, 63], are k-A-complete for fault domains that contain all FSMs in which any state can be reached by first performing a sequence from a state cover A for the specification model, followed by up to k arbitrary inputs, for some small k. This means that when all states of the SUT can be reached by first doing an access sequence for some state of the specification, followed by a few arbitrary inputs, then these MBT approaches are guaranteed to find bugs in this SUT (when present). Outside these fault domains they are much less effective. Indeed, for the benchmarks considered in this paper, k-A-complete approaches

are ineffective, and we did not use them in our final experiments. Our benchmarks have up to 200 inputs, which means that we can only run the MBT algorithms for values of k up to 2. We conjecture that for many of our benchmarks it is not possible to reach the faulty SUT states by an access sequence of a specification DFA state followed by at most 2 arbitrary inputs. Exploring this conjecture (which might explain the limitations of existing MBT algorithms) is a question for future research.

Specifications help to find bugs faster. Our results show that, when BBC is used to detect bugs in implementations, it really helps to have a high-level specification available. Even in cases where model learning is unable to learn the full SUT model, BBC is able to learn a small hypothesis model that enables the detection of specification violations. In one of our benchmarks (RERS M65), for example, the full SUT model has 3,966 states, but hypothesis models with (a mean of) 14 states are enough to detect that the SUT violates 5 out of the 17 properties in the specification. This provides a nice illustration of the aphorism attributed to George Box that "all models are wrong but some are useful."

7 Limitations and Future Work

All benchmark models in our study have been obtained using model learning from real world protocol implementations and real embedded controllers. All specifications for the protocol benchmarks have been derived from the corresponding protocol standards. The LTL specifications for the RERS benchmarks, however, have been artificially generated using a property mining approach [31]. It would be interesting to further explore the effectiveness of BBC to find specification violations in embedded control software.

In our experiments, we restricted ourselves to a single model learning method, a single equivalence oracle, and a single model-based testing approach. We are confident that the main conclusions of Section 6 will also apply for different algorithms for model learning and model-based testing: model checking hypotheses against the specification provides guidance and helps BBC to find bugs faster. Nevertheless, of course, this needs to be confirmed by further studies.

In our study, we used the total number of learning and testing queries, as well as the total number of inputs and resets, to measure the efficiency of bug finding approaches. In applications there is often a strong correlation between the total number of queries and the time spent on testing/learning. However, performing a query with many input symbols will take longer than a query with just a few symbols. Thus, sometimes the total number of input symbols and resets provides a better measure of the efficiency. For certain applications, resets are impossible or extremely time consuming. Groz et al. [27] designed the hW-inference algorithm for learning models using only a *single* query. We expect that it will be rather easy to combine hW-inference and BBC.

Learning-based testing (and thus BBC) belongs to the general area of fuzzing of stateful systems. As observed by [12], "many fuzzers use some form of learning to infer information about the message format, the protocol state machine, or both. Evolution can be regarded as a form of learning because it produces and uses new knowledge about the input format, even though this knowledge is (usually) not expressed in the form of a regular expression, state machine, or context-fee grammar." BBC is a black-box technique but it has been frequently and successfully applied to find bugs in software for which the source code was available. For those applications it would be interesting to compare the effectiveness of BBC with other (grey-box or white-box) fuzzing approaches. Fuzzing tools typically target memory corruption bugs that crash the SUT, which is more restrictive than the main objective of BBC: finding specification violations. However, a recent white-box fuzzing approach that is closely related to our work has been proposed by Asadian *et.al.* [5]. They encode protocol requirements by monitors, and then employ symbolic execution to detect violations of these requirements in protocol implementations.

Learning-based testing, a.k.a. active automata learning, is a vibrant research area that is progressing rapidly, see, e.g., [14, 24, 33]. Primary challenges are to extend the learning algorithms to richer model classes involving data, timing information and nondeterminism, and scaling the model-based testing algorithms to larger models, in particular in the presence of many possible inputs to the SUT. Two main tools that support active automata learning and provide implementations of the most efficient algorithms are LearnLib [33] and AALpy [40]. LearnLib already supports BBC, based on the work of [37], using LTL formulas as specifications. The BBC implementation described in this paper has been developed on top of AALpy, and accepts both LTL formulas and DFAs as specifications. We think both BBC implementations can be improved by supporting richer languages for describing specifications that are closer to standard engineering practices, e.g., inspired by the bug automata syntax of [21] or automatically generated from Behavior-Driven Development (BDD) specifications [64].

The design of active automata learning algorithms is a subfield of the general area of machine learning. Thus far, however, we have not used LLMs in our study of BBC. We see three promising ways in which use of LLMs may potentially leverage the effective application of BBC: (1) Given the ability of LLMs to analyze RFCs (see, e.g., [42]) it would be interesting to use LLMs to automatically extract formal specifications (e.g., LTL formulas or DFAs) from RFCs. (2) LLMs may help with the construction of adaptors, which translate the abstract inputs and outputs from the specification to concrete messages that are accepted by the SUT (and vice versa). Building adaptors requires both an understanding of the software that will be analyzed as well as an understanding of the intended level of abstraction for the models that one would like to learn. (3) LLMs may support the conformance testing of the SUT with respect to hypothesis models [60].

8 Data Availability Statement

The code and data used to obtain the data covered in this paper is publicly and permanently available on Zenodo and accessible via the DOI 10.5281/zenodo.21769659 on http://doi.org/10.5281/zenodo.21769659. The reviewed version of the artifact is [45].

Acknowledgments

We are most grateful to the reviewers for their valuable feedback, which helped us to further improve this paper. This research was supported by NWO project OCENW.M.23.155 "Evidence-Driven Black-Box Checking (EVI)".

References

[1] F. Aarts, B. Jonsson, J. Uijen, and F.W. Vaandrager. 2015. Generating Models of Infinite-State Communication Protocols using Regular Inference with Abstraction. *Formal Methods in System Design* 46, 1 (2015), 1–41. https://doi.org/10.1007/s10703-014-0216-x

[2] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. 2018. Model Learning and Model-Based Testing. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers (Lecture Notes in Computer Science, Vol. 11026)*, Amel Bennaceur, Reiner Hähnle, and Karl Meinke (Eds.). Springer, 74–100. https://doi.org/10.1007/978-3-319-96562-8_3

[3] Bernhard K. Aichernig, Martin Tappler, and Felix Wallner. 2024. Benchmarking Combinations of Learning and Testing Algorithms for Automata Learning. *Formal Aspects Comput.* 36, 1 (2024), 3:1–3:37. https://doi.org/10.1145/3605360

[4] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Information and Computation* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

[5] Hooman Asadian, Paul Fiterau-Brostean, Bengt Jonsson, and Konstantinos Sagonas. 2024. Monitor-based Testing of Network Protocol Implementations Using Symbolic Execution. In *Proceedings of the 19th International Conference on Availability, Reliability and Security, ARES 2024, Vienna, Austria, 30 July 2024 - 2 August 2024*. ACM, 17:1–17:12. https://doi.org/10.1145/3664476.3664521

[6] C. Baier and J.-P. Katoen. 2008. *Principles of Model Checking*. MIT Press, Cambridge, Massachusetts.

[7] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner (Eds.). 2005. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*. Lecture Notes in Computer Science, Vol. 3472. Springer. https://doi.org/10.1007/b137241

[8] Timothy A. Budd and Dana Angluin. 1982. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica* 18 (1982), 31–45. https://doi.org/10.1007/BF00625279

[9] T.S. Chow. 1978. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering* 4, 3 (1978), 178–187. https://doi.org/10.1109/TSE.1978.231496

[10] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2404)*, Ed Brinksma and Kim Guldstrand Larsen (Eds.). Springer, 359–364. https://doi.org/10.1007/3-540-45657-0_29

[11] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). 2018. *Handbook of Model Checking*. Springer. https://doi.org/10.1007/978-3-319-10575-8

[12] Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. 2024. Fuzzers for Stateful Systems: Survey and Research Directions. *ACM Comput. Surv.* 56, 9 (2024), 222:1–222:23. https://doi.org/10.1145/3648468

[13] Joeri de Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the 24th USENIX Security Symposium, USENIX Security 15*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 193–206. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

[14] Simon Dierl, Paul Fiterau-Brostean, Falk Howar, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2024. Scalable Tree-based Register Automata Learning. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 87–108. https://doi.org/10.1007/978-3-031-57249-4_5

[15] Rita Dorofeeva, Khaled El-Fakih, Stéphane Maag, Ana R. Cavalli, and Nina Yevtushenko. 2010. FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Information & Software Technology* 52, 12 (2010), 1286–1297. https://doi.org/10.1016/j.infsof.2010.07.001

[16] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. 2022. From Spot 2.0 to Spot 2.10: What's New?. In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22) (Lecture Notes in Computer Science, Vol. 13372)*. Springer, 174–187. https://doi.org/10.1007/978-3-031-13188-2_9

[17] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. 2021. Prognosis: closed-box analysis of network protocol implementations. In *Proceedings of the ACM SIGCOMM 2021 Conference*, Fernando A. Kuipers and Matthew C. Caesar (Eds.). ACM, 762–774. https://doi.org/10.1145/3452296.3472938

[18] Paul Fiterau-Brostean and Falk Howar. 2017. Learning-Based Testing the Sliding Window Behavior of TCP Implementations. In *Proceedings of the Critical Systems: Formal Methods and Automated Verification - Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems FMICS and 17th International Workshop on Automated Verification of Critical Systems AVoCS 2017 (Lecture Notes in Computer Science, Vol. 10471)*, Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti (Eds.). Springer, 185–200. https://doi.org/10.1007/978-3-319-67113-0_12

[19] Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. 2016. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *Proceedings of the 28th International Conference Computer Aided Verification, CAV 2016 (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 454–471. https://doi.org/10.1007/978-3-319-41540-6_25

[20] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *Proceedings of the 29th USENIX Security Symposium, USENIX Security 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2523–2540. https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean

[21] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2023. Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society. https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_s68_paper.pdf

[22] Paul Fiterau-Brostean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager, and Patrick Verleg. 2017. 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, July 10-14, 2017*, Hakan Erdogmus and Klaus Havelund (Eds.). ACM, 142–151. https://doi.org/10.1145/3092282.3092289

[23] P. Fiterau-Brostean, E. Poll, F.W. Vaandrager, T. Lenaerts, J.E.J. de Ruiter, and P. Verleg. 2018. Source code and data relevant for the paper 'Model Learning and Model Checking of SSH Implementations'. https://doi.org/10.17026/DANS-Z6N-DXQ6

[24] Markus Frohme, Falk Howar, and Bernhard Steffen. 2025. LearnLib: 10 years later. In *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part IV (Lecture Notes in Computer Science)*, Ruzica Piskac and Zvonimir Rakamaric (Eds.). Springer, 141–160. https://doi.org/10.1007/978-3-031-98685-7_7

[25] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. 1991. Test selection based on finite state models. *IEEE Transactions on Software Engineering* 17, 6 (1991), 591–603. https://doi.org/10.1109/32.87284

[26] Bharat Garhewal and Carlos Diego Nascimento Damasceno. 2023. An Experimental Evaluation of Conformance Testing Techniques in Active Automata Learning. In *26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023, Västerås, Sweden, October 1-6, 2023*. IEEE, 217–227. https://doi.org/10.1109/MODELS58315.2023.00012

[27] Roland Groz, Nicolas Brémond, Adenilso da Silva Simão, and Catherine Oriat. 2020. hW-inference: A heuristic approach to retrieve models through black box testing. *J. Syst. Softw.* 159 (2020). https://doi.org/10.1016/j.jss.2019.110426

[28] Richard Hamlet. 1994. Random testing. *Encyclopedia of software Engineering* 2 (1994), 971–978.

[29] J.E. Hopcroft and J.D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

[30] Falk Howar and Bernhard Steffen. 2018. Active Automata Learning in Practice - An Annotated Bibliography of the Years 2011 to 2016. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers (Lecture Notes in Computer Science, Vol. 11026)*, Amel Bennaceur, Reiner Hähnle, and Karl Meinke (Eds.). Springer, 123–148. https://doi.org/10.1007/978-3-319-96562-8_5

[31] Marc Jasper, Malte Mues, Alnis Murtovi, Maximilian Schlüter, Falk Howar, Bernhard Steffen, Markus Schordan, Dennis Hendriks, Ramon R. H. Schiffelers, Harco Kuppens, and Frits W. Vaandrager. 2019. RERS 2019: Combining Synthesis with Real-World Models. In *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 11429)*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer, 101–115. https://doi.org/10.1007/978-3-030-17502-3_7

[32] Loes Kruger, Sebastian Junges, and Jurriaan Rot. 2024. Small Test Suites for Active Automata Learning. In *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 14571)*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer, 109–129. https://doi.org/10.1007/978-3-031-57249-4_6

[33] Loes Kruger, Sebastian Junges, and Jurriaan Rot. 2026. Error-Awareness Accelerates Active Automata Learning. In *Formal Methods - 27th International Symposium, FM 2026, Tokyo, Japan, May 18-22, 2026, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 16556)*, Augusto Sampaio and Marielle Stoelinga (Eds.). Springer, 520–540. https://doi.org/10.1007/978-3-032-26204-2_27

[34] D. Lee and M. Yannakakis. 1996. Principles and methods of testing finite state machines — a survey. *Proc. IEEE* 84, 8 (1996), 1090–1123.

[35] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303. https://doi.org/10.1016/j.jlap.2008.08.004 The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[36] Gang Luo, Alexandre Petrenko, and Gregor von Bochmann. 1995. Selecting Test Sequences for Partially-Specified Nondeterministic Finite State Machines. In *Protocol Test Systems: 7th workshop 7th IFIP WG 6.1 international workshop on protocol text systems*, Tadanori Mizuno, Teruo Higashino, and Norio Shiratori (Eds.). Springer US, Boston, MA, 95–110. https://doi.org/10.1007/978-0-387-34883-4_6

[37] Jeroen Meijer and Jaco van de Pol. 2019. Sound black-box checking in the LearnLib. *Innov. Syst. Softw. Eng.* 15, 3-4 (2019), 267–287. https://doi.org/10.1007/s11334-019-00342-6

[38] Karl Meinke, Fei Niu, and Muddassar A. Sindhu. 2011. Learning-Based Software Testing: A Tutorial. In *Leveraging Applications of Formal Methods, Verification, and Validation - International Workshops, SARS 2011 and MLSC 2011. Revised Selected Papers (Communications in Computer and Information Science, Vol. 336)*, Reiner Hähnle, Jens Knoop, Tiziana Margaria, Dietmar Schreiner, and Bernhard Steffen (Eds.). Springer, 200–219. https://doi.org/10.1007/978-3-642-34781-8_16

[39] Karl Meinke and Muddassar A. Sindhu. 2011. Incremental Learning-Based Testing for Reactive Systems. In *Tests and Proofs - 5th International Conference, TAP@TOOLS 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6706)*, Martin Gogolla and Burkhart Wolff (Eds.). Springer, 134–151. https://doi.org/10.1007/978-3-642-21768-5_11

[40] Edi Muskardin, Bernhard Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. 2022. AALpy: an active automata learning library. *Innovations in Systems and Software Engineering* 18 (03 2022), 1–10. https://doi.org/10.1007/s11334-022-00449-3

[41] Daniel Neider, Rick Smetsers, Frits W. Vaandrager, and Harco Kuppens. 2018. Benchmarks for Automata Learning and Conformance Testing. In *Models, Mindsets, Meta: The What, the How, and the Why Not? (Lecture Notes in Computer Science, Vol. 11200)*, Tiziana Margaria, Susanne Graf, and Kim G. Larsen (Eds.). Springer, 390–416. https://doi.org/10.1007/978-3-030-22348-9_23

[42] Mrigank Pawagi, Lize Shao, Hyeonmin Lee, Yixin Sun, and Wenxi Wang. 2025. RFCScope: Detecting Logical Ambiguities in Internet Protocol Specifications. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1234–1246. https://doi.org/10.1109/ASE63991.2025.00106

[43] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. 1999. Black Box Checking. In *Proceedings FORTE (IFIP Conference Proceedings, Vol. 156)*, Jianping Wu, Samuel T. Chanson, and Qiang Gao (Eds.). Kluwer, 225–240. https://doi.org/10.1007/978-0-387-35578-8_13

[44] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. 2002. Black Box Checking. *J. Autom. Lang. Comb.* 7, 2 (2002), 225–246. https://doi.org/10.25596/jalc-2002-225

[45] Bram Pellen, María Belén Rodríguez, Frits Vaandrager, and Petra van den Bos. 2026. Quick Bug Detection Through Black-Box Checking: a Systematic Evaluation (Artifact). doi:10.5281/zenodo.21389827

[46] Alexandre Petrenko and Nina Yevtushenko. 2006. Conformance Tests as Checking Experiments for Partial Nondeterministic FSM. In *Formal Approaches to Software Testing*, Wolfgang Grieskamp and Carsten Weise (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 118–133. https://doi.org/10.1007/11759744_9

[47] Alexandre Petrenko, Nina Yevtushenko, Alexandre Lebedev, and Anindya Das. 1993. Nondeterministic State Machines in Protocol Conformance Testing. In *Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems, Pau, France, 28-30 September, 1993 (IFIP Transactions, Vol. C-19)*, Omar Rafiq (Ed.). North-Holland, 363–378.

[48] Andrea Pferscher and Bernhard K. Aichernig. 2022. Stateful Black-Box Fuzzing of Bluetooth Devices Using Automata Learning. In *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13260)*, Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer, 373–392. https://doi.org/10.1007/978-3-031-06773-0_20

[49] Harald Raffelt and Bernhard Steffen. 2006. LearnLib: A Library for Automata Learning and Experimentation. In *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science)*, Luciano Baresi and Reiko Heckel (Eds.). Springer, 377–380. https://doi.org/10.1007/11693017_28

[50] M. Schuts, J. Hooman, and F.W. Vaandrager. 2016. Refactoring of Legacy Software using Model Learning and Equivalence Checking: An Industrial Experience Report. In *Proceedings 12th International Conference on integrated Formal Methods (iFM) (LNCS, Vol. 9681)*, E. Ábrahám and M. Huisman (Eds.). 311–325. https://doi.org/10.1007/978-3-319-33693-0_20

[51] Wouter Smeenk, Joshua Moerman, Frits W. Vaandrager, and David N. Jansen. 2015. Applying Automata Learning to Embedded Control Software. In *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, France, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9407)*, Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi (Eds.). Springer, 67–83. https://doi.org/10.1007/978-3-319-25423-4_5

[52] Jan Tretmans. 2008. Model Based Testing with Labelled Transition Systems. In *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, Robert M. Hierons, Jonathan P. Bowen, and Mark Harman (Eds.). Springer, Berlin, Heidelberg, 1–38. https://doi.org/10.1007/978-3-540-78917-8_1

[53] Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. 2022. A New Approach for Active Automata Learning Based on Apartness. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 223–243. https://doi.org/10.1007/978-3-030-99524-9_12

[54] Frits Vaandrager and Ivo Melse. 2025. New Fault Domains for Conformance Testing of Finite State Machines. In *36th International Conference on Concurrency Theory (CONCUR 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 348)*, Patricia Bouyer and Jaco van de Pol (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 34:1–34:22. https://doi.org/10.4230/LIPIcs.CONCUR.2025.34

[55] Frits Vaandrager and Thorsten Wißmann. 2023. Action Codes. In *50th International Colloquium on Automata, Languages, and Programming (ICALP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 261)*, Kousha Etessami, Uriel Feige, and Gabriele Puppis (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 137:1–137:20. https://doi.org/10.4230/LIPIcs.ICALP.2023.137

[56] Frits W. Vaandrager. 2017. Model learning. *Commun. ACM* 60, 2 (2017), 86–95. https://doi.org/10.1145/2967606

[57] Petra van den Bos, Ramon Janssen, and Joshua Moerman. 2019. n-Complete test suites for IOCO. *Softw. Qual. J.* 27, 2 (2019), 563–588. https://doi.org/10.1007/s11219-018-9422-x

[58] Petra van den Bos and Frits W. Vaandrager. 2021. State identification for labeled transition systems with inputs and outputs. *Sci. Comput. Program.* 209 (2021), 102678. https://doi.org/10.1016/j.scico.2021.102678

[59] M.P. Vasilevskii. 1973. Failure Diagnosis of Automata. *Cybernetics and System Analysis* 9, 4 (1973), 653–665. https://doi.org/10.1007/BF01068590 (Translated from Kibernetika, No. 4, pp. 98-108, July-August, 1973.).

[60] Yunze Wei, Kaiwen Wei, Shibo Du, Jianyu Wang, Zhangzhong Liu, Yawen Wang, Zhanyou Li, Congcong Miao, Xiaohui Xie, and Yong Cui. 2025. Automated Network Protocol Testing with LLM Agents. arXiv:2510.13248 [cs.NI] https://doi.org/10.48550/arXiv.2510.13248

[61] Elaine J. Weyuker. 1983. Assessing Test Data Adequacy through Program Inference. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 641–655. https://doi.org/10.1145/69575.357231

[62] Nan Yang, Kousar Aslam, Ramon R. H. Schiffelers, Leonard Lensink, Dennis Hendriks, Loek Cleophas, and Alexander Serebrenik. 2019. Improving Model Inference in Industry by Combining Active and Passive Learning. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 253–263. https://doi.org/10.1109/SANER.2019.8668007

[63] N. V. Yevtushenko and A. F. Petrenko. 1991. Synthesis of test experiments in some classes of automata. *Autom. Control Comput. Sci.* 24, 4 (apr 1991), 50–55.

[64] Tannaz Zameni, Petra van den Bos, Arend Rensink, and Jan Tretmans. 2024. An Intermediate Language to Integrate Behavior-Driven Development Scenarios and Model-Based Testing. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024 - Companion, Rovaniemi, Finland, March 12, 2024*. IEEE, 199–206. https://doi.org/10.1109/SANER-C62648.2024.00033

[65] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2024. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/ Retrieved 2024-07-01 16:50:18+02:00.

Received 2026-03-26; accepted 2026-06-18