Innovations in Systems and Software Engineering (2019) 15:267–287
https://doi.org/10.1007/s11334-019-00342-6

S.I. : NFM2018

Sound black-box checking in the LearnLib

Jeroen Meijer[1] · **Jaco van de Pol**[1,2]

Received: 1 October 2018 / Accepted: 8 May 2019 / Published online: 30 May 2019

Abstract
In black-box checking (BBC) incremental hypotheses on the behavior of a system are learned in the form of finite automata, using information from a given set of requirements, specified in Linear-time Temporal Logic (LTL). The LTL formulae are checked on intermediate automata and potential counterexamples are validated on the actual system. Spurious counterexamples are used by the learner to refine these automata. We improve BBC in two directions. First, we improve checking lasso-like counterexamples by assuming a check for state equivalence. This provides a sound method without knowing an upper-bound on the number of states in the system. Second, we propose to check the safety portion of an LTL property first, by deriving simple counterexamples using monitors. We extended LearnLib's system under learning API to make our methods accessible, using LTSmin as model checker under the hood. We illustrate how LearnLib's most recent active learning algorithms can be used for BBC in practice. Using the RERS 2017 challenge, we provide experimental results on the performance of all LearnLib's active learning algorithms when applied in a BBC setting. We will show that the novel incremental algorithms TTT and ADT perform the best. We also provide experiments on the efficiency of various BBC strategies.

Keywords Black-box checking · LTL · LearnLib · Model checking · Monitors · Learn-based testing · LTSmin · Büchi automata

1 Introduction

There are many formal methods for analyzing the desired behavior of complex industrial critical systems, such as wafer steppers and X-ray diffraction machines. These systems are used in the production and analysis microchips, a multi-billion dollar industry. From a formal methods perspective both liveness (something good eventually happens), and safety (something bad never happens) are essential to the functional reliability of those systems. It is key for testers and developers to have easily usable tooling available to investigate those liveness and safety properties. We expand the formal method toolbox of testers and developers by contributing the first free and open source software (FOSS) implementation of black-box checking (BBC) [32] to the LearnLib.

Supported by STW SUMBAT Grant: 13859. Supported by the 3TU.BSR Project.

✉ Jeroen Meijer
j.j.g.meijer@utwente.nl

Jaco van de Pol
jaco@cs.au.dk

[1] University of Twente, Enschede, The Netherlands

[2] Aarhus University, Aarhus, Denmark

The contributed method is based on learning and is used as a formal approach to testing systems (hence, the method is also known as Learning-Based Testing (LBT) [29]). Here, requirements are checked on a model of the system that is automatically learned from observations. In particular, the requirements are applied to intermediate learned hypotheses in order to speed up the learning process or tested on the system. We implemented BBC in the LearnLib [27], showing its ease-of-use. There we also introduced a state equivalence check, to obtain a sound method without assuming an upper-bound on the number of states in the System Under Learning (SUL). In this extended article, we will describe the method and the design of its implementation in more detail. Additionally, we present a novel method for the safety portion of the requirements by means of so-called monitors. We performed extensive experiments comparing both methods under several BBC strategies, to show how well they perform on an actual case study.

1.1 BBC among other formal methods

Functional requirements document the desired behavior of a system. Following the formal methods approach, these requirements are often formulated with some kind of tempo-

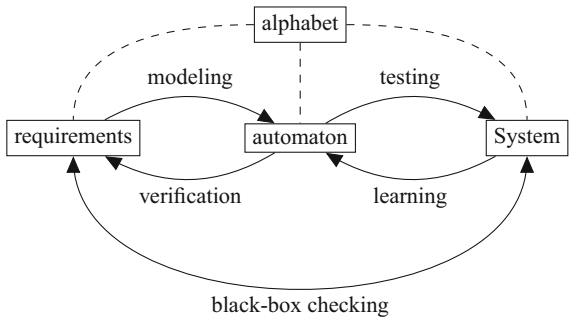


Fig. 1 Example formal methods

ral logic, such as Linear-time Temporal Logic (LTL). In order to relate the system to its requirements we identify four complementary formal methods: *verification*, *testing*, *modeling* and *learning*. To start, we distinguish the System Under Test (SUT) from its automaton-based description. An automaton is a mathematical abstract representation of the behavior of the SUT. *Verification* involves checking whether the automaton is correct with respect to the SUT by means of the formalized requirements. *Testing* involves checking whether the system conforms to the automaton representation of the system. When an automaton is correctly *modeled* according to the SUT, an approach known as Model-Based Testing [43] is typically applied for testing the SUT. As shown in Fig. 1, the behavior of the automaton and the requirements should be kept synchronized, which can be a burden to the developers of the system. So, instead of modeling the automaton by hand, it has been proposed that the automaton can be *learned* automatically from interacting with the SUT. This procedure is called Active Automata Learning (AAL) [38]. LearnLib [20] is a library that contains a wide variety of AAL algorithms. Many of these algorithms are inspired by Angluin's famous L^* algorithm [1].

Figure 1 also shows the concept of an *alphabet*. An alphabet contains the symbols in which requirements must be written, and in what language the system communicates with the environment. This means that to make the system perform an action an input must be sent that is a symbol in the alphabet. To observe the reaction of the system, the output must also be a symbol in the alphabet. The alphabet is a key ingredient that binds the mentioned formal methods together.

Testing, verification, modeling and learning can be used in a complementary fashion, because all of them have their advantages. Verification can be done by means of model checking. Model checking has been around for several decades and efficient model checkers are readily available. The advantage of using formal models for testing is a highly automated approach to check whether the system conforms to the model. There are many mature MBT tools available, such as JTorX [4] that supports a sizable number of input modeling languages such as mCRL2. From a practical perspective, learning an automaton from a system is quite straightforward, because the only requirements are a definition of the alphabet, and some kind of adapter between a learning algorithm and SUT. These adapters are often quite easy to build. The four methods also have disadvantages. For example, when verification is performed, it is known which requirements hold on the model of the system, but due to its abstract nature, it is uncertain which of those requirements also hold on the actual concrete system. Traditionally, model-based testing has the disadvantage that the automaton has to be modeled and maintained by hand. Writing specifications for automata can be tedious, since the domain specific languages (e.g., process algebras, such as mCRL2) may be unfamiliar to the developers of the system. Verifying requirements on an automaton that is obtained through learning is not always feasible either. That is because it can take quite a long time before learning algorithms produce a high enough quality automaton. And even when such an automaton is obtained, verifying requirements is not straightforward, because the learned automaton can still be incorrect. Black-box checking tries to alleviate those problems. It alleviates the need for maintaining an automaton of the system by implicitly learning it, so the user perceives it as if the requirements were directly tested on the system.

In general, the BBC procedure requires checking that an infinite length counterexample can be executed on the SUT. An infinite counterexample is represented by a lasso of the form uv^ω, where u represents the initialization of the system, and v models an infinite loop. These lassos are provided by model checkers that check the automata by means of the LTL properties. When an LTL property can not be verified, the model checker must provide a counterexample in the form of such a lasso. The soundness of the original BBC procedure [32] assumed guessing an upper-bound on the number of states in the system. This could be either dangerous (if the guess is too low), or inefficient (if the guess is too high). We resolved this in [27] by allowing the LearnLib to check for state equivalence in the SUL by implementing so-called ω-queries. In this work, we present a novel approach that uses monitors. The advantage of this new approach is that when a system cannot be correctly monitored, a finite counterexample is produced. Validating such a finite counterexample from a model checker by simply testing whether the SUL accepts it is inherently sound. For safety properties, soundness is preserved without the need of checking for state equivalence. For arbitrary LTL properties, we propose to first check if their "safety portion" can be used, before resorting to the general procedure using lasso-like counterexamples.

1.2 Contributions

To summarize the contributions; we will revisit the work done on lasso-shaped counterexamples in [27]. Additionally,

we cover the concept of monitoring and how this is implemented in the LearnLib. The various AAL algorithms and BBC strategies that are now available are subjected to rigorous experiments. Concretely we contribute the following.

- Two variations of black-box checking algorithms.
- A sound black-box checking approach that uses state equivalences, instead of an upper-bound on the number of states in the SUL.
- A novel sound black-box checking approach that uses monitors that may provide counterexamples for safety properties.
- A modular design, allowing new model checkers or active learning algorithms to be added easily, or smarter strategies to be implemented for detecting spurious counterexamples.
- A thorough reproducible experimental setup, with several combinations of automaton types, AAL algorithms and BBC strategies.

The rest of the paper is structured as follows. Section 2 provides preliminary definitions and procedures for LTL model checking, active learning and black-box checking. Section 3 describes monitoring, how one can check whether a SUL accepts an infinite lasso-shaped word, and how this is implemented in the LearnLib. In Sect. 4 we discuss related work, such as other model checkers, active learning algorithms, and BBC approaches. Section 5 details the result of our case study, and Sect. 6 concludes our work and discusses possibilities for future research.

2 Preliminaries

The LearnLib mainly contains AAL algorithms for Deterministic Finite Automata (DFAs) and Mealy machines. We provide a definition for both, and a definition for Labeled Transition Systems (LTSs) where multiple labels per edge are allowed. Typically, model checkers verify LTL properties on LTSs. Hence we provide LTL semantics for LTSs, and provide straightforward translations from DFAs and Mealy machines to LTSs. Implementations of these translations have been added to the LearnLib. Furthermore, this section gives a short introduction to active learning, and black-box checking.

Definition 1 (*Edge Labeled Transition System*) An edge Labeled Transition System (LTS) is defined as a tuple $\mathscr{L} = \langle S, s_0, \delta, L, T, \lambda \rangle$, where

- S is a finite nonempty set of states,
- $s_0 \in S$ is the initial state,
- $\delta : S \to 2^S$ is the transition function,
- L is the set of edge labels,
- T is the set of edge label types, and
- $\lambda : S \times S \times T \to L$ is the edge labeling function.

We use the shorthand notation $\lambda(s, s') = \{(t, \lambda(s, s', t)) \mid t \in T\}$ to obtain all edge labels of a transition. Furthermore, a *path* in $\mathscr{L}$ is an infinite sequence of states beginning in s_0. The set of paths is $\text{Paths}(\mathscr{L}) = \{s_0 s_1 \ldots \in S^\omega \mid \forall i > 0 \colon s_i \in \delta(s_{i-1})\}$. A *trace* is an infinite sequence of sets of tuples of labels:

$$\text{Traces}(\mathscr{L}) = \{\lambda(s_0, s_1)\lambda(s_1, s_2) \ldots \in (2^{T \times L})^\omega \mid s \in \text{Paths}(\mathscr{L})\}.$$

The set of *finite prefixes* of $\mathscr{L}$ is:

$$\text{Pref}(\mathscr{L}) = \{p \in (2^{T \times L})^* \mid \exists s \in (2^{T \times L})^\omega \colon ps \in \text{Traces}(\mathscr{L})\}.$$

Note that as a consequence of the definition of $\text{Traces}(\mathscr{L})$, prefixes that lead to a deadlock state are *not* in $\text{Pref}(\mathscr{L})$. Consequently, prefixes that lead to a deadlock state can never serve as counterexamples to LTL properties.

Definition 2 (*Deterministic Finite Automaton*) A Deterministic Finite Automaton (DFA) is defined as a tuple $\mathscr{D} = \langle S, s_0, \Sigma, \delta, F \rangle$, where

- S is a finite nonempty set of states,
- $s_0 \in S$ is the initial state,
- Σ is a finite alphabet,
- $\delta : S \times \Sigma \to S$ is the *total* transition function,
- $F \subseteq S$ is the set of accepting states.

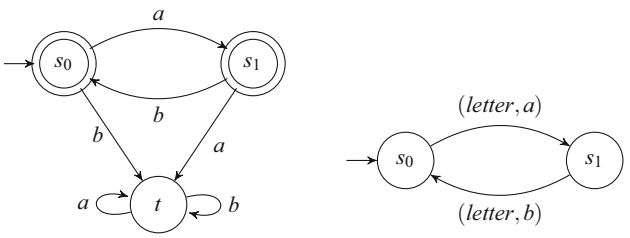
The language of $\mathscr{D}$ is denoted $L(\mathscr{D})$. A DFA is *Prefix-Closed* iff $\forall s \in S, \forall i \in \Sigma \colon \delta(s, i) \in F \implies s \in F$. This implies that $\forall \sigma_1 \ldots \sigma_n \in L(\mathscr{D}) \colon \sigma_1 \ldots \sigma_{n-1} \in L(\mathscr{D})$. The LTS of a nonempty, prefix-closed DFA $\mathscr{D}$ is $\mathscr{L}_\mathscr{D} = \langle F, s_0, \delta_\mathscr{L}, \Sigma, \{letter\}, \lambda_\mathscr{L} \rangle$, where

- $\delta_\mathscr{L}(s) = \bigcup_{i \in \Sigma} \delta(s, i)$, and
- $\lambda_\mathscr{L}(s, s') = \{(letter, l) \mid l \in \Sigma \wedge \delta(s, l) = s'\}$.

Example 1 (DFA) An example prefix-closed DFA for the regular expression $(ab)^*(a?)$ is given in Fig. 2a. The LTS is given in Fig. 2b. This LTS has only a single trace, $\{(letter, a)\}\{(letter, b)\} \cdots$. The set of finite prefixes is the set $\{\{(letter, a)\}, \{(letter, a)\}\{(letter, b)\}, \ldots\}$.

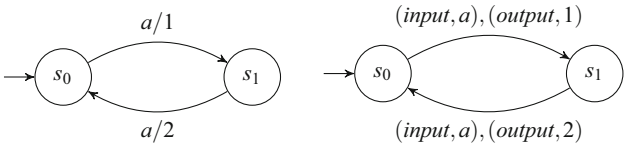
Definition 3 (*Mealy Machine*) A Mealy machine is defined as a tuple $\mathscr{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$, where

- S is a finite nonempty set of states,



(a) DFA **(b)** LTS

Fig. 2 Example DFA



(a) Mealy machine **(b)** LTS

Fig. 3 Example Mealy machine

- $s_0 \in S$ is the initial state,
- Σ is a finite input alphabet,
- Ω is a finite output alphabet,
- $\delta : S \times \Sigma \to S$ is the *total* transition function, and
- $\lambda : S \times \Sigma \to \Omega$ is the *total* output function.

The LTS of $\mathscr{M}$ is $\mathscr{L}_{\mathscr{M}} = \langle S, s_0, \delta_{\mathscr{L}}, \Sigma \cup \Omega, \{input, output\}, \lambda_{\mathscr{L}} \rangle$, where

- $\delta_{\mathscr{L}}(s) = \bigcup_{i \in \Sigma} \delta(s, i)$, and
- $\lambda_{\mathscr{L}}(s, s') = \{\{(input, i), (output, o)\} \mid i \in \Sigma \wedge \delta(s, i) = s' \wedge o \in \Omega \wedge \lambda(s, i) = o\}$.

Example 2 (*Mealy Machine*) An example Mealy machine is given in Fig. 3a. The LTS is given in Fig. 3b. The only trace of this LTS is: $\{(input, a), (output, 1)\}\{(input, a), (output, 2)\}\cdots$.

Throughout this paper, the following assumptions are made.

- We only consider DFAs that reject the empty language (otherwise their LTS is not defined).
- We only consider prefix-closed DFAs (Mealy machines are by definition prefix-closed).
- We only consider minimal DFAs and Mealy machines (automata constructed through active learning are always minimal; our definition of prefix-closed only holds on minimal automata).
- We assume that the SUL is deterministic.

2.1 LTL model checking

An LTL formula expresses a property that should hold over all infinite runs of a system. This means that if a system does not satisfy an LTL property, there generally exists a counterexample that is an infinite word which exhibits a lasso structure.

Definition 4 (*LTL*) Given an LTS $\mathscr{L} = \langle S, s_0, \delta, L, T, \lambda \rangle$, LTL formulae over $\mathscr{L}$ adhere to the following grammar:[1]

$$\phi ::= \texttt{true} \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \texttt{X}\,\phi \mid \phi_1 \,\texttt{U}\, \phi_2 \mid t = l,$$

where $t \in T$, and $l \in L$. Given an LTL formula ϕ, all infinite words that satisfy ϕ are given by the set $\text{Words}(\phi) = \{\sigma \in (2^{T \times L})^\omega \mid \sigma \models \phi\}$, where the satisfaction relation $\models \subseteq (T \times L)^\omega \times \text{LTL}$ is defined inductively over ϕ by the following properties. Let $\sigma = A_0 A_1 A_2 \ldots \in (2^{T \times L})^\omega$, and $\sigma[j] = A_j A_{j+1} A_{j+2} \ldots$:

$$\sigma \models \begin{cases} \texttt{true}, & \\ \phi_1 \wedge \phi_2 & \text{iff } \sigma \models \phi_1 \wedge \sigma \models \phi_2, \\ \neg\phi & \text{iff } \sigma \not\models \phi, \\ \texttt{X}\,\phi & \text{iff } \sigma[1] \models \phi, \\ \phi_1 \,\texttt{U}\, \phi_2 & \text{iff } \exists j : \sigma[j] \models \phi_2 \wedge \forall i < j : \sigma[i] \models \phi_1, \\ t = l & \text{iff } (t, l) \in A_0. \end{cases}$$

Furthermore, the set of *finite prefixes* of the words that satisfy ϕ is: $\text{Pref}(\phi) = \{p \in (2^{T \times L})^* \mid \exists s \in (2^{T \times L})^\omega : ps \in \text{Words}(\mathscr{L})\}$. We say that $\mathscr{L}$ *satisfies* ϕ iff $\text{Traces}(\mathscr{L}) \subseteq \text{Words}(\phi)$, and that ϕ *monitors* $\mathscr{L}$ iff $\text{Pref}(\mathscr{L}) \subseteq \text{Pref}(\phi)$. Both "$\mathscr{L}$ satisfies ϕ" and "ϕ monitors $\mathscr{L}$" can be checked with model checkers, so for clarity we refer to them as procedures named *model checking* and *monitoring*, respectively. Note that for safety properties, $\mathscr{L}$ satisfies ϕ if and only if ϕ monitors $\mathscr{L}$. For arbitrary LTL properties we only have implication. So we can use monitoring to check at least the "safety portion" of ϕ.

The practical advantage of using monitoring instead of model checking is that monitoring provides finite counterexamples, whereas model checking requires infinite counterexamples, here represented as lassos.

Definition 5 (*Lasso*) Given an LTS $\mathscr{L}$, a trace $\sigma \in \text{Traces}(\mathscr{L})$ is a *lasso* if it can be split in a finite prefix p, such that $p \sqsubset \sigma$, and a finite *loop* q, such that $pq^\omega = \sigma$.

Example 3 (LTL for DFAs) An example LTL formula that is satisfied by the LTS $\mathscr{L}$ in Fig. 2b is: $\phi = \texttt{X}(letter = b)$. All the words that satisfy the formula are in $\text{Words}(\phi) = \{\{(letter, a)\}\{(letter, b)\}\ldots, \{(letter, b)\}\{(letter, b)\}\ldots\}$.

Clearly, $\text{Traces}(\mathscr{L}) \subseteq \text{Words}(\phi)$, so $\mathscr{L}$ satisfies ϕ (and also ϕ monitors $\mathscr{L}$, because $\text{Pref}(\mathscr{L}) \subseteq \text{Pref}(\phi)$). Note that ϕ is a safety property, and if we make a small change:

[1] Extensions and equivalences may be defined as in [3] (such as implication: $\Longrightarrow$, globally: G, and future: F).

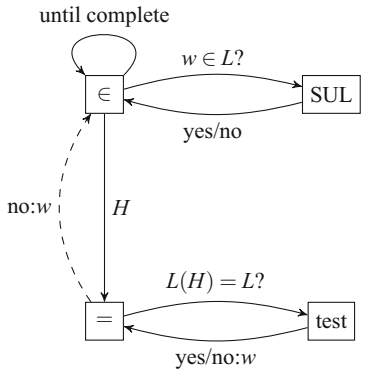
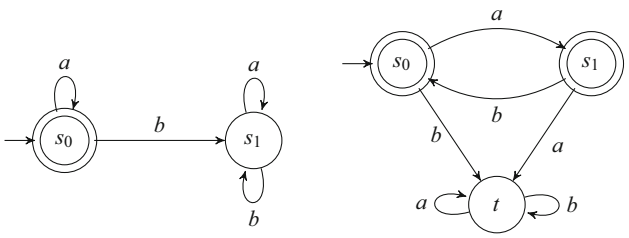


Fig. 4 Active learning procedure



(a) First Hypothesis **(b)** Final hypothesis

Fig. 5 Active learning

$\phi' = \mathtt{X}(letter = a)$, it can not be monitored. A *finite* counterexample then is $\{(letter, a)\}\{(letter, b)\}$. A lasso to show that ϕ' can not be satisfied is $\epsilon(\{(letter, a)\}\{(letter, b)\})^{\omega}$.

An example for Mealy machines is analogous. For checking whether a formula ϕ monitors, or is satisfied by an LTS $\mathscr{L}$, a traditional model checker can be used. It will perform the emptiness check $\text{Pref}(\mathscr{L}) \cap \overline{\text{Pref}(\phi)} = \varnothing$ and the emptiness check $\text{Traces}(\mathscr{L}) \cap \overline{\text{Words}(\phi)} = \varnothing$, respectively. Following the automata-based approach, this is done by computing Cartesian product of their respective automata representations. A monitor automaton can recognize $\text{Pref}(\phi)$ and a Büchi automaton can recognize $\text{Traces}(\phi)$. Note that instead of performing emptiness checks, one can also perform the original inclusion check ($\text{Pref}(\mathscr{L}) \subseteq \text{Pref}(\phi)$) by checking the invariant that all reachable states in the monitor are accepting. This approach requires that the monitor is deterministic. We used LTSmin [22] to implement the inclusion check.

2.2 Active learning

For our purposes, active learning is the process of learning a sequence of hypotheses $\mathscr{H}_1\mathscr{H}_2 \ldots \mathscr{H}_F$, such that their behavior converges to some target automaton (DFA, or Mealy machine). The key components are illustrated in Fig. 4.

Learner an algorithm that can form hypotheses based on queries and counterexamples. A learning algorithm will pose queries until it has *complete* information to form a hypothesis. The notion of completeness depends on the applied learning algorithm, e.g., the L* algorithm will perform queries until its observation table is closed and consistent.

Equivalence oracle (=) an oracle that decides whether two languages are equal. The oracle decides between the language of the current hypothesis of the learner and the language of the SUL. If the languages are not equivalent the oracle will provide a counterexample that distinguishes both languages. Here, the language of the SUL is a set of finite traces. Note that the distinguishing word is either in the language of the hypothesis, or the language of the SUL, but not both.

Membership oracle (∈) an oracle that decides whether or not a word is a member of the language of the SUL.

SUL In the case, an active learning algorithm is applied to an actual system, a SUL interface is used that can *step* through a system in order to answer membership queries. In the LearnLib, the `SUL` interface exposes the methods `pre` and `post` that can reset a system (i.e., put it back to the initial state), `step` that stimulates the system with one input symbol and returns the corresponding output, `canFork` and `fork` that may fork a SUL, i.e., provide some copy (that behaves identically to) the forked system. In active learning, this is used to pose queries in parallel. We will show its usefulness for performing state equivalence checks in the context of BBC too. More information about the `SUL` interface can be found in Fig. 8.

Example 4 (Active Learning) Given an alphabet $\Sigma = \{a, b\}$, and a DFA $\mathscr{D}$ to be learned such that $L(\mathscr{D}) = (ab)^*a?$, an active learning algorithm could first produce the hypothesis $\mathscr{D}_1$ in Fig. 5a, where the language accepted is $L(\mathscr{D}_1) = \mathrm{a}^*$. At some point the equivalence oracle generates $aa \in \Sigma^*$, and performs the membership query $aa \in \mathscr{L}?$ = no. The equivalence oracle recognizes that $aa \in L(\mathscr{D}_1)$, and concludes it found a counterexample to $\mathscr{D}_1$. The learner refines $\mathscr{D}_1$, and produces the final hypothesis in Fig. 5b and we are done learning. Note that this example hides the complexity of actually refining the hypothesis. In the LearnLib refining a hypothesis is done with the method `Learner#refineHypothe-sis()` that accepts a query (counterexample) and subsequently poses additional membership queries. More details on refining hypotheses are outside of the scope of this paper; they can be found in, e.g., [1,38].

Finding a counterexample to the current hypothesis by means of an equivalence oracle can be very time-consuming. In the worst-case, the equivalence oracle has to try out all words of maximum length N in $\Sigma^{0\ldots N}$. Some smart equivalence oracles (e.g., ones using the partial W-method [12]) can find a counterexample quite quickly, if there is one. However,

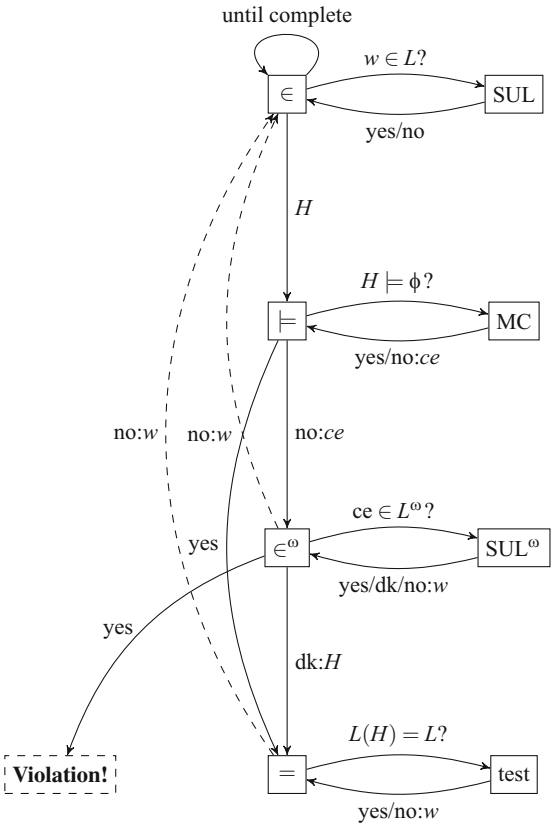


Fig. 6 Sound black-box checking procedure [33]

the number of membership queries to find the counterexample is still orders of magnitudes larger than the size of the hypothesis. For example, any word of maximum length $N = 2$ that could serve as a counterexample for the first hypothesis in Example 4 is in $\{\epsilon, a, b, aa, ab, ba, bb\}$. When hypotheses grow larger, the set of possible counterexamples grows with an even larger degree. Black-box checking alleviates this problem by using LTL properties to restrict the search space of such counterexamples.

2.3 Black-box checking with model checking

The sound approach to black-box checking is illustrated in Fig. 6; it builds upon the active learning algorithm. The difference, however, is that hypotheses are not immediately forwarded to an equivalence oracle, but first subjected to a model checker.

When the model checker provides a counterexample (ce) to the property, this is tested on the SUL. If the counterexample can be simulated on the system, we found a violation of the property. If not, a prefix of the counterexample (w) is provided to the learner, saving one expensive test-procedure.

A complication is that the counterexample provided by the model checker is an infinite trace, presented by a lasso xy^ω. In principle, one can only check finite unrollings xy^n on the system. However, this yields an unsound method, unless one knows an upper-bound on the number of states of the SUL.

An initial sound approach to black-box checking was proposed in [27]. Membership of infinite words ($\in^\omega$) is checked, by assuming that one can additionally save states and check their equivalence. So we test the word and save intermediate states $x(s_0)y(s_1)y(s_2), \ldots, y(s_n)$. As soon as we find that $s_k = s_j$ for some $0 \leq k < j \leq n$, we definitely know that xy^ω is a valid counterexample, and report a violation. If the path cannot be continued, we have a found a finite prefix w for the learner. Otherwise, we don't know (dk) if xy^ω holds, and we proceed to the tester.

The adapted procedure is sound, in the sense that it only reports true violations. However, it may miss some violations, so it is incomplete: first, the final hypothesis may still not reflect all system behavior. Second, the model checker may have detected a lasso that could not be confirmed within the bound.

This work presents an additional sound approach in Sect. 3.4. Before checking whether lassos are accepted, we first apply monitoring. If monitoring reveals that the safety portion of the property does not hold on the hypothesis, there exists a finite counterexample that needs to be confirmed on the SUL. Naturally, this is much more practicable than having to check lasso-like counterexamples for every property.

Example 5 (Black-Box Checking) Consider again the first hypothesis $\mathscr{D}_1$, produced by an active learning algorithm from Fig. 5a, that accepts the language a^*, and the LTL formula $\phi = \mathtt{X}(letter = b)$, from Example 3. An LTL model checker checks whether the LTS of $\mathscr{D}_1$ satisfies formula ϕ ($\mathscr{L}_{\mathscr{D}_1} \models \phi$). The model checker concludes $\mathscr{L}_{\mathscr{D}_1}$ does not satisfy ϕ, and produces the lasso a^ω as a counterexample. The lasso from the model checker is verified on the SUL by means of the membership oracle that can perform omega queries, i.e., $\in^\omega$ performs the query $a^\omega \in L^\omega$? = no. This means that with hypothesis $\mathscr{D}_1$ the property can not be disproved. Let us assume that $\in^\omega$ unrolled a^ω twice when performing the membership query, then $aa \notin L(\mathscr{D}_1)$ is provided as a counterexample to the learner. In practice, the number of times the loop of the lasso is unrolled depends on the size of the hypothesis. The essence of the current example is that Fig. 5a can be refined without performing any equivalence query. This example (like Example 4 about active learning) hides the complexity of refining a hypothesis too. Refining a hypothesis in the LearnLib in the context of BBC can also be done with `Learner#refineHypothesis()`.

3 Sound black-box checking in the LearnLib

This section provides the detailed description and some extensions of our sound BBC approach introduced in [27],

including the BBC algorithm and its integration in the LearnLib [20]. The main new contributions are an extension of sound BBC with finite counterexamples from monitors, the description of various strategies to interleave property checking and hypothesis refinement, and a detailed overview of the design in LearnLib's API.

3.1 Black-box checking in the LearnLib

We implemented a more general procedure for black-box checking in the LearnLib than presented in Sect. 2.3. First, the result from the model checker is generalized to a language representing a set of counterexamples to a property, instead of just a single word. This follows from our observation that some counterexamples returned by the model checker are uninformative or spurious, since hypotheses may be incorrect. So instead of a single membership query to validate counterexamples from the model checker an *emptiness oracle* checks whether the intersection between the counterexample language returned by the model checker and the language of the SUL is empty. If it is not empty, an example in the intersection serves as a counterexample to the property. Secondly, instead of checking just a single property, the LearnLib can check a set of properties. This check is implemented as a loop, in which more and more properties are disproved. Having to check a set of properties gives rise to multiple strategies. One strategy tries to disprove as many properties as possible, before providing a counterexample to refine the current hypothesis. The other strategy tries to disprove a single property; if it can not be disproved it tries to refine the current hypothesis, before continuing with the next property. In Sect. 3.6 we will detail how the user of the LearnLib can pick either of those two strategies, and in Sect. 5, we will investigate the efficiency of these strategies. The following components are added to the LearnLib:

Model checker ($\models$) An algorithm that checks whether a property is satisfied or is monitored by a hypothesis. If the check fails the result is a set of counterexamples, which is in fact a subset of the language of the checked hypothesis.

Emptiness oracle ($\varnothing$) An oracle that decides whether the intersection of two languages is empty. The oracle decides between the language of the counterexamples given by the model checker and the language of the SUL. If the intersection is not empty it will provide a counterexample, which is a word in the intersection and as such, a counterexample to the property checked by the model checker.

Inclusion oracle ($\subseteq$) An oracle that decides whether one language is included in another. The oracle decides whether the language of the counterexamples given by the model checker is included in the language of the SUL. If the language is not included, the oracle will provide a counterexample outside the language of the SUL, and thus a counterexample to the current hypothesis. One can view the combination of the *model checker*, *emptiness oracle*, and *inclusion oracle* as a *property oracle*, and a set of property oracles as a *black-box oracle*.

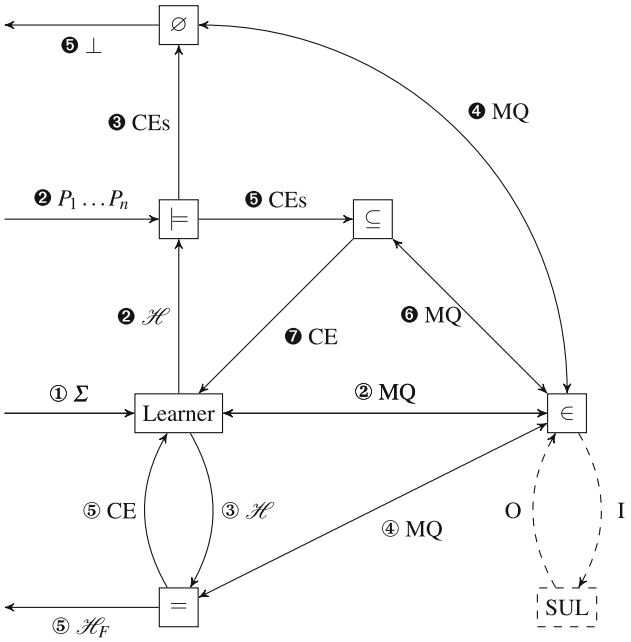


Fig. 7 Black-box checking algorithm in the LearnLib

3.2 New purposes for queries

In traditional active learning, there are two kinds of sets of membership queries; *learning queries* (done by the learner) and *equivalence queries* (done by the equivalence oracle). With BBC, there are two more types of queries; *inclusion queries* (done by the inclusion oracle), and *emptiness queries* (done by the emptiness oracle). The decision between performing inclusion queries and emptiness queries depends on whether the property can be falsified with the current hypothesis. We generalize both to *model checking queries*. The key observation why adding properties to verify to the learning algorithm can be useful, follows from the observation that model checking queries are very cheap compared to equivalence queries. Given an alphabet Σ, a naive equivalence oracle has to perform arbitrary membership queries for words in Σ^*, while the black-box oracle has to perform only membership queries for a subset of the language of the current hypothesis.

3.3 The BBC algorithm and strategies: informally

Now, given that model checking queries are much cheaper than equivalence queries we present the black-box checking algorithm in the LearnLib in Fig. 7. Note that black numbers (e.g., ❷) represent black-box checking steps, and white

numbers (e.g., ②) active learning steps. Also note that steps can have multiple alternatives (e.g., ❺). The BBC procedure is now as follows. Initially (①) the learner constructs an hypothesis using membership queries (②). This hypothesis is, together with a set of properties, given to the model checker (❷). If the model checker finds counterexamples for a property and the current hypothesis, the counterexamples are given to the emptiness oracle (❸). The emptiness oracle performs membership queries (❹) to try to find a counterexample from the model checker that is not spurious. If a real counterexample for a property is found, it is reported to the user (❺), and the property is not considered for future hypotheses. Otherwise, to identify spurious counterexamples, the set of counterexamples is forwarded to the inclusion oracle. The inclusion oracle performs membership queries (❻) to find a counterexample for the current hypothesis (❼). This is given back to the learner, which continues performing membership queries (②) to complete the next hypothesis. If the hypothesis is refined, the black-box oracle repeats steps (❷,…,❼) until the model checker can not find any new counterexample. In the latter case, we enter the traditional active learning loop (Fig. 4): the equivalence oracle tries to find a counterexample for the current hypothesis (③) using membership queries (④). If a counterexample is found (⑤) the learner will construct the next hypothesis using membership queries (②) and the black-box oracle is put back to work. If the equivalence oracle does not find a counterexample (④) the final hypothesis is reported to the user. We can now better illustrate the two black-box oracle strategies. Either, the black-box oracle can first try to find a counterexample for every property before finding a refinement for the current hypothesis. Or, it finds a counterexample for a single property and if such a counterexample does not exist, search for a counterexample to the current hypothesis, before checking the next property. The first implementation is more efficient if there is a high chance a property can be disproved with the current hypothesis, or if refining the current hypothesis becomes quite expensive.

3.4 Black-box checking with monitoring

In traditional active learning the concept of a *query* provides the main interface between the learner or equivalence oracle and the SUL. In this work, the emptiness oracle also uses queries to validate counterexamples from the model checker. A *query* is denoted as an input word $q \in \Sigma^*$ that can be answered by a membership oracle.

Definition 6 (*Membership oracle*) Given a set of queries Q, the set of Booleans $\mathbb{B} = \{\bot, \top\}$, and a SUL S, a *membership oracle* is a function $\in\colon Q \to \mathbb{B}$, such that $\in(q) = q \in L(S)$. A membership oracle for Mealy machines can be defined similarly, see [36].

Example 6 (Answering a query) Consider the example safety property $\phi = (letter = b)$. Then ϕ does not hold on the LTS $\mathscr{L}$ of the final DFA in Fig. 5b. When the model checker checks whether ϕ monitors $\mathscr{L}$ it could provide the singleton language $\{a\}$ as a counterexample to ϕ. The emptiness oracle will ask the membership oracle to answer the query $q = a$ so that the counterexample can be validated. Since a is accepted by the SUL (i.e., $a \in L(S)$) the membership oracle will answer $\in(q) = \top$, and the emptiness oracle will report the answered query as a valid counterexample to ϕ.

Practically, and in case of monitoring, the product of an LTS $\mathscr{L}$ and a monitor automaton of $\overline{\text{Pref}(\phi)}$ is computed, while checking for a witness that visits an accepting state once. Finding such a witness can be achieved on-the-fly with any reachability algorithm. In this work, the automaton accepting the language of the formula is created by Spot [10], while computing the product with the LTS and search for witness is done by LTSmin.

3.5 Black-box checking with model checking

In case we want to know whether an LTS satisfies a formula ϕ, the product of an LTS and a Büchi automaton of $\overline{\text{Words}(\phi)}$ is computed, while checking for a witness that visits an accepting state in the Büchi automaton infinitely often. Searching for such a witness can be done on-the-fly [9] with (concurrent) nested depth-first search [8,25] or SCC-based approaches [5,42].

Making the BBC procedure sound involves checking whether infinite lasso-shaped words given as counterexamples by the model checker are accepted by the SUL. Obviously, checking whether a SUL accepts an infinite word in practice is impossible. However, this can be resolved if one considers what goes on inside a black-box system. We need to check if the SUL also exhibits a particular lasso through its state space when stimulated with a finite word (that also produces the same output as given by the model checker). This can be achieved by observing particular states the SUL evolves through when stimulated. Note that this view of a SUL is still quite a black-box view; we only record the states, we do not *enforce* the SUL to move to a particular state.

We introduce a new notion of a *query*, named an ω-*query*, which in addition to the input word and output of the SUL also contains the periodicity of the loop of the lasso. An ω-query serves as the interface between the emptiness oracle and SUL.

Definition 7 (ω-*query*) Given an alphabet Σ, an ω-*query* is a tuple $q^\omega = (p, l, r) \in \Sigma^* \times \Sigma^+ \times \mathbb{N}$, where

- p is the *prefix* of the lasso to check,
- l the *loop* of the lasso to check,

– $r \geq 1$ the maximum number of times the loop may be unrolled.

Following Definition 6 an ω-membership oracle is used to answer ω-queries and is defined as follows.

Definition 8 (ω-*membership oracle*) Given a set of ω-queries Q^ω over an alphabet Σ, a SUL S with a set of internal states Z, and a function $\text{State}_S \colon \Sigma^* \to Z$, such that $\text{State}_S(\sigma) = z$ gives the internal state z in S after input σ, an ω-*membership oracle* is a function $\in^\omega \colon Q^\omega \to \mathbb{N}$, such that $\in^\omega(q^\omega = (p, l, r)) = n$, where $r \geq n$ indicates the periodicity of l such that $n > 0 \iff \exists i < n \colon \text{State}_S(pl^i) = \text{State}_S(pl^n) \wedge pl^n \in L(S)$. We can also illustrate an answered ω-query with $n > 0$ as

$$\xrightarrow{p} \cdot \overbrace{\underbrace{\xrightarrow{l} \cdot \ldots \cdot \xrightarrow{l} s}_{i} \xrightarrow{l} \cdot \ldots \cdot \xrightarrow{l}}^{n} s', \quad \text{where } s = s'.$$

So in case $n > 0$ we generalize $pl^n \in L(S)$ to $pl^\omega \in L^\omega(S)$, because $pl^i(l^{n-i})^\omega = pl^\omega$. A definition for an ω-membership oracle for Mealy machines is similar. For learning a Mealy machine with output alphabet Ω an ω-membership oracle is a function $\in^\omega \colon Q^\omega \to \Omega^+ \times \mathbb{N}$, such that $\in^\omega(q^\omega = (p, l, r)) = (o, n)$ and iff $n > 0$, o is the output string of input pl^n. In Sect. 3.6, we will explain how the above state equivalence check is implemented in the LearnLib. We will also detail how checking for state equivalence is done *on-the-fly* when states can be serialized.

Example 7 (*answering an* ω-query) An example property that does not hold for on the LTS $\mathscr{L}$ of the final DFA in Fig. 5b is $\phi = (letter = b)$. Whenever a model checker determines whether $\mathscr{L}$ satisfies ϕ, it may give the lasso $\ell = a(ba)^\omega$ as a potential counterexample to ϕ. The language $\{\ell\}$ is given to the emptiness oracle, with a limit to unroll the loop at most 3 times for instance, and asks the ω-membership oracle to answer the query $q^\omega = (a, ba, 3)$. When stimulating the SUL with $a(ba)^1$, it becomes clear the SUL cycles through state s_1. Hence, the ω-membership oracle will answer $\in^\omega(q^\omega) = 1$. From the answer to the query, it becomes clear that that the SUL accepts the infinite lasso-shaped word ℓ with periodicity 1, and the emptiness oracle will report the answered ω-query as a valid counterexample to ϕ. In practice, the LearnLib will determine the maximum number of unrolls relative to the number of states in the hypothesis automaton on which ϕ is checked.

3.6 The new API in the LearnLib

We extend the API (Application Programmer's Interface) of the LearnLib following Fig. 7. Among others, we add the concept of a model checker ($\models$) as an interface named `ModelChecker` and similar goes for inclusion oracles ($\subseteq$) and emptiness oracles ($\varnothing$). Other new first-class citizens are derived from Definitions 7 and 8 and named accordingly; these establish the form of communication between the emptiness oracle and system. This form of communication is similar to the interface between the learner and system in the original AAL setting.

LearnLib's API extension is illustrated in a class diagram in Fig. 8. Note that for illustration purposes we do not show association arrows between classes, instead associations are represented as class attributes. If an association has a multiplicity greater or equal to zero, we suffix the attribute with the array notation "`[]`". Furthermore, all attributes of a class indicate required parameters of its constructor. A description of the new API is as follows.

`ObservableSUL`: The `SUL` interface is extended with methods `getState() : S` returning the current state of the SUL, `boolean deepCopies()` indicating whether the object returned by `getState()` is a deep copy, and a refinement of `fork()`.

`ModelChecker`: A `ModelChecker` may find a counterexample to a property and hypothesis. The counterexamples are finite words and are a subset of the language of the hypothesis. LTSmin [5,22] is an available implementation of a `ModelChecker` for Monitors in the LearnLib. A `ModelCheckerLasso` is a refinement of a `ModelChecker` that uses Büchi automata and where the counterexamples are lassos instead of finite words. The implementations are named `LTSminMonitor` and `LTSminLTL`, respectively.

`OmegaQuery`: An `OmegaQuery` following Definition 7 is similar to a `Query`. An `answered` `OmegaQuery` stores information about whether an infinite word is in the language of the SUL. Note that type `D` denotes the output *domain* type for different automaton types that are learned.

`EmptinessOracle`: This oracle generates words that are in a given automaton, and tests whether those words are also in the SUL. The implementation `BFEmptinessOracle`, generates words in a breadth-first manner. A limit can be placed on the maximum number of words by supplying a *multiplier* that is used to limit the number of queries performed. This limit is computed by multiplying the size of the given automaton by the specified *multiplier*. An `EmptinessOracle` is used by `PropertyOracles` to check whether any word in the language given as a counterexample by the `ModelChecker` is present in the SUL. A specialization of an `EmptinessOracle` is a `LassoEmptinessOracle` that uses `OmegaQueries` to check whether infinite lasso-shaped words are not in the SUL.

`InclusionOracle`: Similar to the `EmptinessOracle`; it generates a limited number of words in a breadth-first manner, but checks whether words are *in* the language of the SUL. Note that both of these oracles may perform

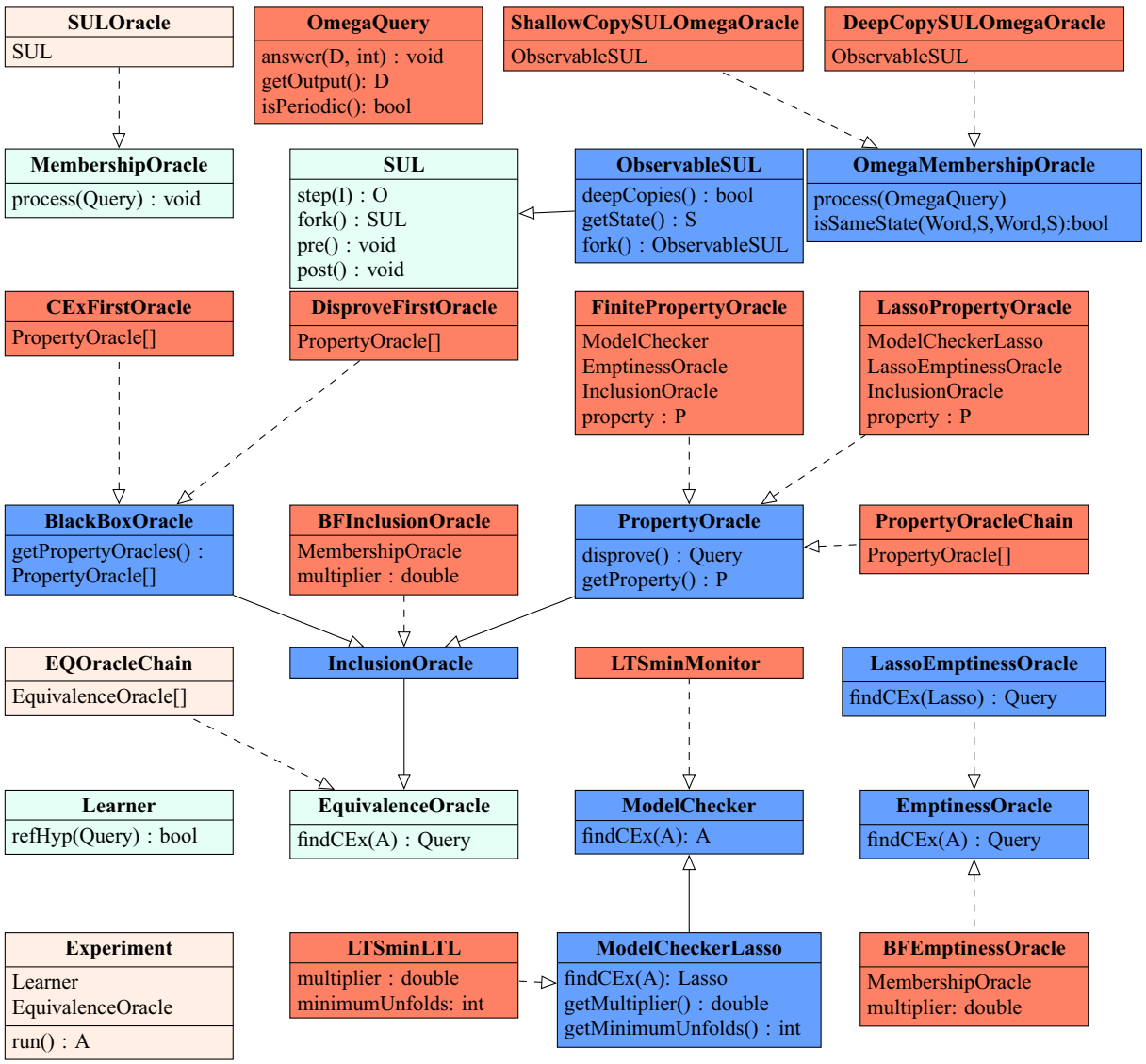


Fig. 8 LearnLib API: classes are red, interfaces are blue, BBC extensions are darker. Solid arrows depict interface refinements, and dashed arrows implementations (color figure online)

the same queries; this is a practical issue and is usually resolved by using a `SULCache` so that in case of a cache-hit the SUL is not stimulated. The `InclusionOracle`, and `EmptinessOracle` may have different strategies (BFS vs. DFS), and hence are not merged together into a single oracle. Separation of concerns (finding a counterexample to the current hypothesis, vs. finding a counterexample to a property), is also considered a good design principle.

`PropertyOracle`: A `PropertyOracle` is an oracle for a property of a black-box system. Such an oracle tries to *disprove* the property or finds a *counterexample* to the current hypothesis. To this end, implementations require a `ModelChecker`, `EmptinessOracle`, `InclusionOracle`, and the property itself. In case the property should be model checked with a monitor one should construct a `FinitePropertyOracle`, in case the property should be checked with a Büchi automaton one should construct a `LassoPropertyOracle`.

`BlackBoxOracle`: An oracle that disproves a set of properties by means of multiple `PropertyOracles` or finds a counterexample to the current hypothesis in the same collection of `PropertyOracles`. Currently, there are two implementations available.

1. `DisproveFirstOracle`: Iterates over the set of properties that are still unknown, and tries to disprove all of them before refining the current hypothesis.
2. `CExFirstOracle`: This implementation iterates over the set of properties that are still unknown, and before

disproving a next property it first tries to refine the current hypothesis with the current property.

Both implementations execute a loop, trying to disprove as many properties as possible. The pseudocode of these strategies is provided in Sect. 3.7. Both implementations will be evaluated later by the experiments in Sect. 5.

`OmegaMembershipOracle`: An oracle that decides if an infinite word is in the language of the SUL, see Definition 8. To this end it poses `OmegaQueries`. There are several implementations available; one that simulates DFAs and Mealy machines directly, and one that wraps around an `ObservableSUL`, by means of `ShallowCopySULOmegaOracle` and `DeepCopySULOmegaOracle`. Based on the implementation of `ObservableSUL#deepCopies()` one can decide to construct a `ShallowCopySULOmegaOracle` or a `DeepCopySULOmegaOracle`. If an `ObservableSUL` does not make a deep copy of the state of the SUL it could be the case that when `SUL#step()` is executed, a previously obtained state with `ObservableSUL#getState()` would also be modified, e.g., the assertion in the following Java snippet may not hold.

```
1 ObservableSUL oSUL = ...;
2 S s = oSUL.getState();
3 int hc = s.hashCode();
4 oSUL.step(...);
5 assert s.hashCode() == hc;
```

To resolve this; if `ObservableSUL#deepCopies()` does not hold, then `SUL#forkable()` must hold. `ShallowCopySULOmegaOracle` will use two instances of a `ObservableSUL`, i.e., one regular instance, and a `fork`ed instance to compare two states. More specifically, a `ShallowCopySULOmegaOracle` in fact uses hash codes of states, and if the hash codes of two states are equal, then it will step one instance of the `ObservableSUL` through the access sequence of one state, and the `fork`ed instance of the `ObservableSUL` through the access sequence of the second state. When both `ObservableSUL`s are in the desired state `OmegaMembershipOracle#isSameState()` is issued to check if the queried word is in fact a lasso.

In case `ObservableSUL#deepCopies()` *does* hold, checking equality of two states is straightforward. `DeepCopySULOmegaOracle` simply invokes `Object.equals()` on the two states.

Concluding this section; we want to show how conveniently one can set up a BBC experiment with the class `Experiment`. This class needs to be constructed with a `Learner` implementation (e.g., TTT, ADT, etc.) And a *chain* of `EquivalenceOracles`. In our experiment, this chain has two elements: first a `BlackBoxOracle` and then a more complete `EquivalenceOracle`, for instance one that applies the W-method or tries random words. Listing 1 shows exactly how the running example can be implemented in the LearnLib using the chain of `EquivalenceOracles`. In Sect. 5 we show how one can learn a Mealy machine by implementing LearnLib's `SUL` interface.

3.7 The algorithms: formally

Listings 1 to 3 shows how one can set up black-box checking experiments in the LearnLib. The code illustrates how one can setup monitoring with the `CExFirstOracle`, and how to setup model checking with the `DisproveFirstOracle`. The code `Experiment#run()` in Listings 1 and 2 will run exactly the algorithm presented in Algorithm 1, while `Experiment#run()` in Listings 1 and 3 runs exactly the algorithm presented in Algorithm 2. Generics are omitted to improve the presentation of the code listings. The input to both algorithms is a set of LTL properties, an alphabet, and membership oracle(s). Additionally, Algorithm 1 requires a *multiplier* to restrict the maximum number of unrolls of each lasso. Note that when we write $|H|$, we mean the size of automaton H, i.e., the number of states in H. The output of both algorithms is the final hypothesis learned. Additionally, both algorithms will report which properties have been disproved, and what the counterexamples to those properties are.

To be more complete, we compare the original sketch of the BBC algorithm in Fig. 7 to executions of Algorithm 2 (Algorithm 1 is simply a variant). To start the *Learner* component uses the alphabet (①) and membership oracle ($\in$) to initialize the first hypothesis by answering membership queries (②) on line 2. The main loop on line 3 covers all steps in Algorithm 2 except returning the final hypothesis (⑤). In the main loop, we iterate over (line 5) every property (❷) that has not been disproved. This second iteration over properties consists of three parts. The first part at line 6 involves checking whether a property (ϕ) can be monitored by means of the model checker ($\models$). If the property can not be monitored a nonempty language (i.e., action ❸ and variable T) is provided in which every word is a counterexample to ϕ. The second part (lines 7–11) covers the emptiness check that involves the emptiness oracle ($\varnothing$) to disprove properties. The role of the emptiness oracle is to decide no word in T is accepted by the system (❹). When this can not be established (the intersection of T and the language of the system is not empty) a counterexample q is reported as a counterexample to ϕ. This is illustrated as output of the algorithm in step ❺. The third part (lines 12-15) covers the alternative to step ❺ where the emptiness of the intersection is established. The inclusion check involves the inclusion oracle ($\subseteq$) and its role is to disprove hypotheses. To this end, it decides whether

Listing 1 API usage skeleton for black-box checking

```
1         // define the alphabet
2  Alphabet sigma = Alphabets.characters('a', 'b');
3         // create the running example DFA
4  DFA dfa = AutomatonBuilders.newDFA(sigma).
5      withInitial("q0").withAccepting("q0").withAccepting("q1").
6      from("q0").on('a').to("q1").from("q1").on('b').to("q0").create();
7         // create an omega membership oracle, that simulates the DFA
8  OmegaMembershipOracle oMO = new SimulatorOmegaOracle(dfa);
9         // create a regular membership oracle
10 MembershipOracle mO = oMO.getMembershipOracle();
11        // create an equivalence oracle that uses the partial W-method
12 EquivalenceOracle eqO = new WpMethodEQOracle(3, mO);
13        // create a TTT learner
14 Learner learner = new TTTLearnerDFA(sigma, mO, LINEAR_FWD);
15        // create an inclusion oracle, with multiplier 1.0
16 InclusionOracle inO = new DFABFInclusionOracle(mO, 1.0);
17
18              /*************************************************************/
19              /*** insert code here from one of the next two Listings ***/
20              /*************************************************************/
21
22        // modify the equivalence oracle by prepending the black-box oracle
23 eq0 = new EQOracleChain(bbo, eq0);
24        // create an experiment
25 Experiment e = new Experiment(learner, eqO, sigma);
26        // run the experiment
27 DFA finalHypothesis = e.run();
28        // assert we have the correct result
29 assert findSeparatingWord(dfa, finalHypothesis, sigma) == null;
```

Listing 2 API usage for black-box checking with *model checking*, and the `DisproveFirstOracle` strategy

```
1         // create a parser that translates data between LTSmin and the
              LearnLib
2  Function<String, Character> s2c = s -> s.charAt(0);
3         // create an LTSmin model checker
4  ModelCheckerLasso checker = new
       LTSminLTLDFABuilder().withString2Input(s2c).create();
5         // create an emptiness oracle for lassos
6  LassoEmptinessOracle emO = new DFALassoEmptinessOracle(oMO);
7         // create the black-box property from the running example
8  PropertyOracle po = new DFALassoPropertyOracle("X letter==\"b\"", inO, emO,
       checker);
9         // create the black-box oracle with the singleton set of properties
10 BlackBoxOracle bbo = new DisproveFirstOracle(po);
```

every word in T is included in the language of the system by means of membership queries (❻). When however, there is a word ($q \in T$) that is not included in the language of the system, the hypothesis is disproved and is refined with q at line 14 and illustrated in step ❼. The remainder is the traditional equivalence check illustrated with steps ③–⑤ and executed at lines 17–19. This check is executed when there is no property that is able to disprove the current hypothesis.

The crucial difference between Algorithm 1 and 2 is the concept of a `CExFirstOracle` versus a `DisproveFirstOracle`. The first oracle admits an immediate alternative at step ❺. At this step, if a property is disproved but no counterexample is accepted by the system, the language (i.e., the set of counterexamples) is given to the inclusion oracle which will find a counterexample to refine the hypothesis. A `DisproveFirstOracle` on the other hand will first try to disprove every property before refining the hypothesis. Experimental results later show that it is better to use a `DisproveFirstOracle` since it reduces the number of membership queries posed by the learner (②).

Listing 3 API usage for black-box checking with *monitoring*, and the `CExFirstOracle` strategy

```
        // create a parser that translates data between LTSmin and the
           LearnLib
Function<String, Character> s2c = s -> s.charAt(0);
        // create an LTSmin model checker
ModelChecker checker = new
    LTSminMonitorDFABuilder().withString2Input(s2c).create();
        // create an emptiness oracle
EmptinessOracle emO = new DFABFEmptinessOracle(mO);
        // create the black-box property from the running example
PropertyOracle po = new DFAFinitePropertyOracle("X letter==\"b\"", inO, emO,
    checker);
        // create the black-box oracle with the singleton set of properties
BlackBoxOracle bbo = new CExFirstOracle(po);
```

Algorithm 1: Black-box checking with *model checking*, `DisproveFirstOracle`, TTT, and partial W-method

Input : set of LTL properties P, alphabet Σ, membership oracle $\in$, ω-membership oracle $\in^\omega$, and multiplier M
Output: final hypothesis H

$P' \leftarrow \varnothing$ ▷ initialize previous set
$H \leftarrow$ TTT- INIT$(\Sigma, \in)$ ▷ initialize hypothesis with the TTT learning algorithm
while $P' \neq P$ **do** ▷ least fixed-point loop
  $P' \leftarrow P$ ▷ save set of properties to prove
  for $\phi \in P$ **do** ▷ try to disprove ϕ
    $\Omega \leftarrow \text{Traces}(\mathscr{L}_H) \cap \overline{\text{Words}(\phi)}$ ▷ model check ϕ
    for $pq^\omega \in \Omega$ **do** ▷ do emptiness check
      for $1 \leq i \leq |H| * \text{M}$ **do** ▷ compute number of unrolls i
        $q^\omega \leftarrow (p, q, |H| * \text{M})$ ▷ create the ω-query
        if $\in^\omega(q^\omega) > 0$ **then** ▷ answer q^ω and test if it is ultimately periodic
          REPORT(ϕ, pq^ω) ▷ report pq^ω as a counterexample to ϕ
          $P \leftarrow P \setminus \{\phi\}$ ▷ remove ϕ so that it is not checked again
          goto Algorithm 5 ▷ continue with next property
  for $\phi \in P$ **do** ▷ try to disprove H
    $\Omega \leftarrow \text{Traces}(\mathscr{L}_H) \cap \overline{\text{Words}(\phi)}$ ▷ model check ϕ
    for $pq^\omega \in \Omega$ **do** ▷ do inclusion check
      $q \leftarrow pq^{|H|*\text{M}}$ ▷ create the query
      if $\in(q) = \bot$ **then** ▷ answer q and test if q can refine H
        $H \leftarrow$ TTT- REFINE$(H, \Sigma, \in, q, \bot)$ ▷ refine H with q
        goto Algorithm 3 ▷ continue with the next hypothesis
$(q, b) \leftarrow$ Wp-EQUIV$(H, \Sigma, \in)$ ▷ perform an equivalence check with the partial W-method
if $b \neq q \in L(H)$ **then** ▷ test if query q with answer b can refine H
  $H \leftarrow$ TTT- REFINE$(H, \Sigma, \in, q, b)$ ▷ refine H with q and b
  goto Algorithm 3 ▷ continue with the next hypothesis
return H

4 Related work

Related work can be found in several areas. First, there is related work on BBC itself: [27,30,34]. Second, we mention monitoring [41], adaptive learning [13,17] and learning ω-regular languages [2,26] as related research directions with a different focus. Third, other than the LearnLib there is another active learning framework called libalf [6]. Finally, aside from LTSmin there are other model checkers such as NuSMV [7], and SPIN [14]. Readers interested more in AAL are referred to an extensive review by Howar and Steffen [16].

The work in this paper is an extension of [27]. There we extended and implemented the seminal black-box checking approach by [32]. We introduced the notion of ω-query based on loop detection, to obtain soundness of black-box checking. We also experimented extensively with BBC in the context of the more recent algorithms ADT and TTT. Here, we improve our method by first checking the safety part of the LTL properties using monitors before using Büchi automata, to circumvent expensive loop checking when possible. We also added extensive experiments, comparing the approaches using monitors and using Büchi automata. Also, we added experiments to evaluate different strategies on interleaving property checking and hypothesis refinement. Finally, we incorporated more technical details, like algorithms in pseu-

Algorithm 2: Black-box checking with *monitoring*, `CExFirstOracle`, TTT, and partial W-method

Input : set of LTL properties P, alphabet Σ, membership oracle $\in$
Output: final hypothesis H

1 $P' \leftarrow \varnothing$ ▷ initialize previous set
2 $H \leftarrow$ TTT- INIT$(\Sigma, \in)$ ▷ initialize hypothesis with the TTT learning algorithm
3 **while** $P' \neq P$ **do** ▷ least fixed-point loop
4 $P' \leftarrow P$ ▷ save set of properties to prove
5 **for** $\phi \in P$ **do** ▷ iterate over all properties
6 $T \leftarrow \text{Pref}(\mathscr{L}_H) \cap \overline{\text{Pref}(\phi)}$ ▷ monitor ϕ
7 **for** $q \in T$ **do** ▷ do emptiness check
8 **if** $\in(q) = \top$ **then** ▷ answer query q and test if q is a counterexample to ϕ
9 REPORT(ϕ, q) ▷ report q as a counterexample to ϕ
10 $P \leftarrow P \setminus \{\phi\}$ ▷ remove ϕ so that it is not checked again
11 **goto** Algorithm 5 ▷ continue with next property
12 **for** $q \in T$ **do** ▷ do inclusion check
13 **if** $\in(q) = \bot$ **then** ▷ answer q and test if q can refine H
14 $H \leftarrow$ TTT- REFINE$(H, \Sigma, \in, q, \bot)$ ▷ refine H with q
15 **goto** Algorithm 3 ▷ continue with the next hypothesis
16 $(q, b) \leftarrow$ Wp-EQUIV$(H, \Sigma, \in)$ ▷ perform an equivalence check with the partial W-method
17 **if** $b \neq q \in L(H)$ **then** ▷ test if query q with answer b can refine H
18 $H \leftarrow$ TTT- REFINE$(H, \Sigma, \in, q, b)$ ▷ refine H with q and b
19 **goto** Algorithm 3 ▷ continue with the next hypothesis
20 **return** H

docode and an overview of the software design, integrating LTSmin into the LearnLib for BBC.

The concept of *monitoring* in this work is similar to the concept of monitoring in the field of Runtime Verification (RV). In RV a monitor deadlocks when an illegal action may not be performed by the system. Thus in RV, a monitor prevents the system from performing an illegal action. We build *complete* monitors that explicitly reject finite traces if infinite extensions of those traces would violate the original formula. In our work a monitor does not prevent a system from performing an illegal action, we use monitors to check if there are illegal (finite) traces in the hypothesis.

BBC is not entirely new to the LearnLib; several years ago a similar study was performed, named *dynamic testing* [34]. However, since then new active learning algorithms such as ADT [11], and TTT [19] have been added to the LearnLib. Their performance in the context of BBC was still unknown and was first investigated in [27]. Both ADT and TTT are comparable to the Incremental Kripke Learning algorithm (IKL) [29] in LBTest, which is a so-called incremental learning algorithm. Incremental learning algorithms try to produce new hypotheses more quickly, in order to reduce the number of learning queries. Traditional active learning algorithms, such as L* produce fewer hypotheses, but each new hypothesis requires more learning queries. The latter makes sense in the context of active learning, because this minimizes the number of equivalence queries necessary. In the context of active learning, incremental learning algorithms may actually degrade performance; while they may perform well in the number of learning queries, they may require more equivalence queries to refine the hypotheses, resulting in longer run times, see [18, Section 5.5]. In BBC, model checking queries can be used to refine hypotheses. Model checking queries induce negligible overhead compared to equivalence queries [29], making the ADT, and TTT algorithms excellent candidates for a BBC study.

Adaptive learning [13,17] is another paradigm that tries to improve the efficiency and applicability of active automata learning. Adaptive learning is suitable in the context of regression testing, where subsequent versions of the System under Learning are similar. Adaptive learning tries to reuse the model of the previous system, in order to boost learning the new system.

Active learning for ω-regular languages is a related topic, but it has a different focus. There the goal is to learn automata to recognize ω-regular languages, based on ω-queries. Several representations have been suggested, in particular ultimately periodic words [26] and *families of DFAs* [2].

The main reason for our technology choices is related to the availability of the underlying software tools. Currently, LBTest is not free and open source software (FOSS). The LearnLib on the other hand is licensed under the Apache 2 license and thus freely available, even for commercial use. This is interesting because BBC is very successful when applied to industrial critical systems [24,28]. Our new implementation in the LearnLib is also licensed under the Apache 2 license. The reason to implement BBC in the LearnLib instead of libalf is that LearnLib is actively maintained, while libalf is not.

We have chosen to select the LTSmin [22] model checker, because LTSmin, similar to the LearnLib has a liberal BSD

license, and is still actively maintained. For Mealy machines LTSmin implements both *synchronous* and *alternating* trace semantics, which is detailed in [33]. In this work (in particular Sect. 2), we cover the synchronous trace semantics, because these are more intuitive than alternating trace semantics. Compared to NuSMV, LTSmin has an explicit-state model checker, while NuSMV is a symbolic model checker using BDDs. In principle, NuSMV would also suffice as a model checker in this work. We have designed our BBC approach in such a way that in the future integrating NuSMV with the LearnLib is easy; one can simply implement the `ModelChecker` interface. Another popular model checker is SPIN. A disadvantage of using the SPIN model checker is that the counterexamples it produces are state-based, while active learning algorithms require action-based counterexamples [37].

5 Experimental results

BBC in the presence of a good amount of LTL formulae can greatly reduce the number of learning and equivalence queries required to disprove the LTL formulae compared to active learning. In this section, we will quantify the improvement by an experiment. Note that, although BBC introduces additional model checking queries (performed by the emptiness oracle or inclusion oracle), these model checking queries are dwarfed by the amount of equivalence and learning queries. We will thus refrain from reporting the amount of model checking queries here (they can be found online[2], alongside reproduction instructions). What we will show is the amount of learning queries and hypothesis refinements required by various learning algorithms and BBC strategies to disprove as many LTL formulae as possible. We have to be selective in displaying result graphs, since their possible variation is enormous, due to our modular design.

5.1 Variables, metrics and constants

The variables of our experiment (Learning algorithm, BBC strategy and Automaton type) are as follows.

- Eight learning algorithms: ADT [11], DHC [31], Discrimination Tree [15], L* [1], Kearns and Vazirani [23], Maler and Pnueli [26], Rivest and Schapire [35], and TTT [19].
- Three black-box checking algorithms: `CExFirstOracle`, `DisproveFirstOracle`, and – none –.
- Three automaton types: monitor, Büchi automaton, and both monitor and Büchi automaton

[2] https://github.com/Meijuh/NFM-ISSE-2018.

Note that the black-box checking algorithm – *none* – computes the final hypothesis before checking any properties. So this strategy represents a traditional active learning experiment, in which properties are only checked once the final hypothesis has been learned.

To measure the performance of the above variations, we measure the following each time a property is disproved.

1. The number of states in the hypothesis.
2. The number of queries in total.
3. The number of learning queries.
4. The number of equivalence queries.
5. The number of emptiness queries.
6. The number of inclusion queries.
7. The aggregated length of each of above query types, i.e., the total number of symbols.
8. The length of the counterexample that disproves the property.
9. The number of times the hypothesis has been refined.

The constants in the experiments are nine RERS problems with each 100 LTL formulae. Those RERS problems will be introduced in Sect. 5.2. Typically half of the 100 LTL formulae can be falsified. In total we have $(8 * 3 * 3 =)$ 96 variations, and $(96 * 9 =)$ 864 experiments (in a single experiment all LTL properties are checked). These experiments are run on the University of Twente CTIT compute cluster with a time-out of one hour.

The variations and metrics allow us to answer the following research questions.

1. What is the best *learning* algorithm applied in the context of black-box checking? This question will be answered by showing which learning algorithm performs the least amount of learning queries.
2. What is the best *black-box checking* strategy? (i.e., `DisproveFirstOracle` or `CExFirstOracle`) This question will also be answered by showing which black-box checking algorithm performs the least amount of learning queries.
3. What is the *length* of counterexamples when model checking and monitoring? Since, in case of monitoring, there is no need to unroll loops of lassos, counterexamples to properties are expected to be much shorter. We will compare lengths of counterexamples to properties in scatter plots.
4. Do we see a *difference* in performance between classical learning algorithms like L* and modern *incremental* learning algorithms like TTT?
5. How do both model checking and monitoring *affect learning performance*? Model checkers will give different counterexamples to properties for both procedures. Furthermore, in case of monitoring, counterexamples to

properties are much shorter. When these counterexamples to properties are spurious (i.e., they can be used to refine hypotheses) the performance of learning algorithms is affected. To investigate how the performance is affected, we will also show the number of learning queries performed for each.

5.2 The RERS challenge

The Rigorous Examination of Reactive Systems (RERS) challenge[3] is a yearly recurring verification challenge [21]. There are two main categories in this challenge. In one category one has to solve properties for problems which are parallel in nature [40]. The other category involves sequential problems [39]. We are interested in these sequential problems. The RERS sequential problems are provided in Java (among others); the Java problem structure is given in Listing 4.

Listing 4 RERS problem structure

```
@EqualsAndHashCode(exclude = {"inputs"})
public class Problem {
    ...
    public String[] inputs = {"B","E","C","A","D"};

    private int a175 = 6;
    private int a52 = 9;
    private int a176 = 7;
    private String a166 = "e";
    private String a167 = "e";
    private String a62 = "f";

    public String calculateOutput(String i) {...}

    public void reset() {...}
    ...
}
```

The implementation of the `ObservableSUL` interface called `ProblemSUL` is shown in Listing 5. Class `ProblemSUL` connects the RERS problem instances to the LearnLib.

Listing 5 RERS SUL implementation

```
public class ProblemSUL implements ObservableSUL {
    final private Problem problem = new Problem();

    @Override
    public void pre() {
    }

    @Override
    public void post() {
        problem.reset();
    }

    @Override
    public String step(String input) {
        return problem.calculateOutput(input);
    }

    @Override
    public boolean canFork() {
        return true;
    }

    @Override
    public ObservableSUL fork() {
        return new ProblemSUL();
    }

    @Override
    public boolean deepCopies() {
        return false;
    }

    @Override
    public Problem getState() {
        return problem;
    }}
```

One can see that it is straightforward to actively learn a Mealy machine from a `Problem` instance. The *input alphabet* is specified with the field `String[] inputs` in Listing 4. The *state* of a problem instance is determined by the valuations of some instance variables (`a175`, `a52`, `a176`, `a166`, `a167`, and `a62`). An *input* can be given to the `calculateOutput` method, which returns an *output* `String`. The problem instance can be *reset* with the `reset()` method. A SUL implementation of a RERS `Problem` is easy: `SUL#post()` resets the problem instance, and `SUL#step()` applies one input to the problem instance.

With model checking we must be able to retrieve the current state of a `Problem` instance, hence `ProblemSUL` implements `ObservableSUL` instead of only `SUL`. We choose not to make deep copies of a state of a `Problem`, hence `ObservableSUL#deepCopies()` does not hold. We have made this decision, because stepping through a problem instance is cheaper than serializing an entire problem state, which would be necessary if `deepCopies()` would return `true`. A serialized state for larger problem instances could be as big as several kilo bytes. Because we have decided `deepCopies()` returns `false`, an `OmegaMembershipOracle` implementation, must use `Problem#hashCode()`, and `Problem#equals()`. These methods are generated with project Lombok[4], by annotating the `Problem` class with `@EqualsAndHashCode`. Lastly, the `ObservableSUL` can be `fork`ed by creating a new `ProblemSUL` instance. Note that `ObservableSUL` mandates that it must be *fork*able if `Problem` instances are not deep copies.

[3] http://rers-challenge.org.

[4] https://projectlombok.org.

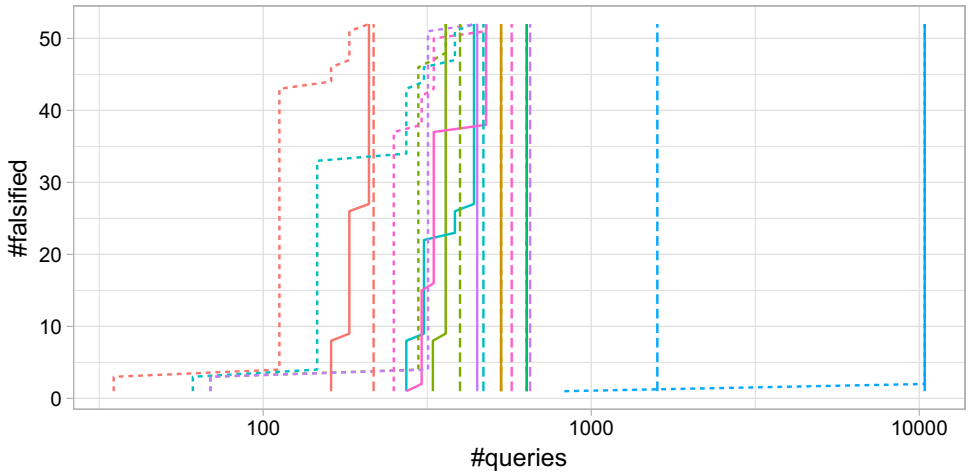


Fig. 9 Number of *learning* queries for *model checking* (with Büchi automata) on problem instance 1, comparing various AAL algorithms and BBC strategies (color figure online)

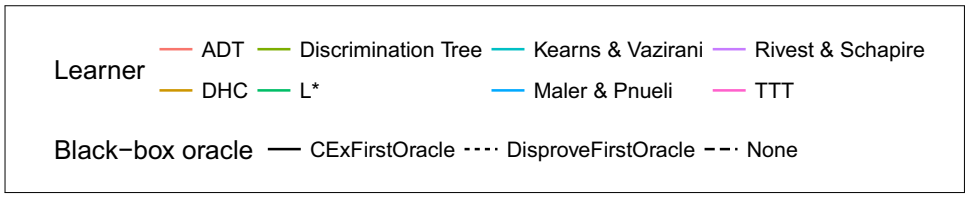


Fig. 10 Legend for learning algorithms and black-box checking algorithms (color figure online)

5.3 Discussion of the algorithms' performance

We can now answer the first two research questions, i.e., what the best learning and black-box checking algorithm are. We will illustrate the performance of these algorithms with cactus plots. The best algorithms for model checking can be read off from Fig. 9.

The figure shows the number of learning queries on RERS problem instance 1 only. The x-axis shows the number of queries required to disprove a certain number of properties. The y-axis shows the amount of properties that are disproved. The line color differentiates the learning algorithms and the type of line differentiates the black-box checking strategies. A legend of these colors and line types is shown in Fig. 10. The further a line appears to the left; the better the algorithms. Figure 11 displays the results for monitoring, but on the larger RERS problem instance 5; it shows which learning algorithms and strategies are superior for the monitoring case. Figure 12 shows the number of hypothesis refinements required for monitoring on the same problem. Note however, that a better algorithm in Fig. 12 does not necessarily appear further to the left. The number of equivalence queries when using BBC algorithms is zero for smaller RERS problems, hence we do not show such a graph. In fact, we would not be able to show such a graph, because the x-axes on the cactus plots shown have logarithmic scales. Lines with *Black-box oracle=none* are always only vertical, because active learning algorithms do not disprove properties on-the-fly (i.e., the same number of queries is required to disprove all properties).

In the cases where BBC algorithms are applied, properties are disproved on-the-fly and result in lines that are strictly monotonic increasing. Interestingly, Fig. 9 shows almost all algorithms use fewer learning queries to disprove some properties when used in the context of BBC. Figure 12 also shows that (as suspected) the incremental TTT, and ADT algorithms ultimately produce more hypotheses than classical algorithms like L* and Rivest and Schapire. This observation answers research question three.

The performance of the eight algorithms is quite consistent throughout the larger problem instances. The ADT algorithm seems to perform really well, but the TTT algorithm is quite competitive too. This is most visible in the larger RERS problems. The great performance of ADT is particularly interesting, due to it being recently developed for AAL — not for BBC. The ADT algorithm is developed to reduce the number of resets of the SUL and is seen as an improvement over TTT. Now, ADT seems to be the best choice for BBC too, among the benchmarked algorithms and RERS problem instances.

Continuing with the fourth research question we will show the length of counterexamples and their differences between model checking and monitoring. Figure 13 shows the difference in length of counterexamples between those produced with Büchi automata and monitors. If a circle appears below the line $x = y$ than the counterexample with monitoring is shorter. However, if a dot appears below the line $y = 0$ the property could not be disproved with monitors. Counterexamples produced with monitors are almost always *much*

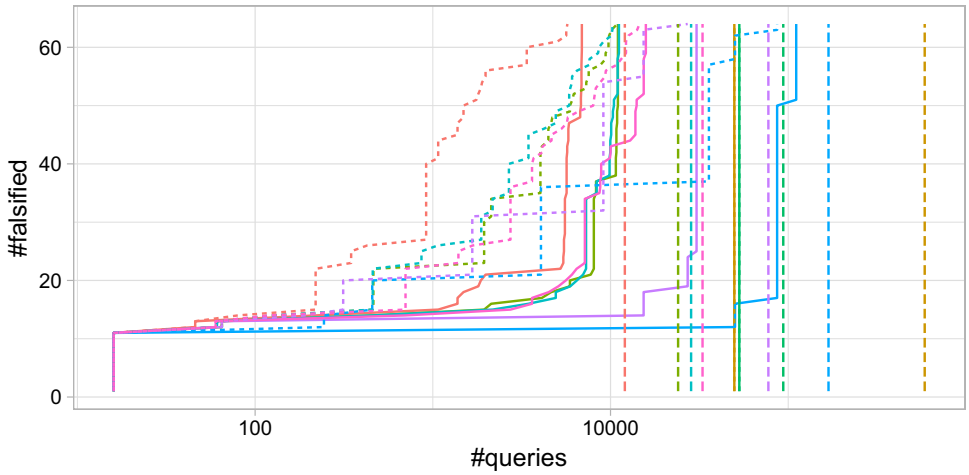


Fig. 11 Number of *learning* queries for *monitoring* on problem instance 5, comparing various AAL algorithms and BBC strategies (color figure online)

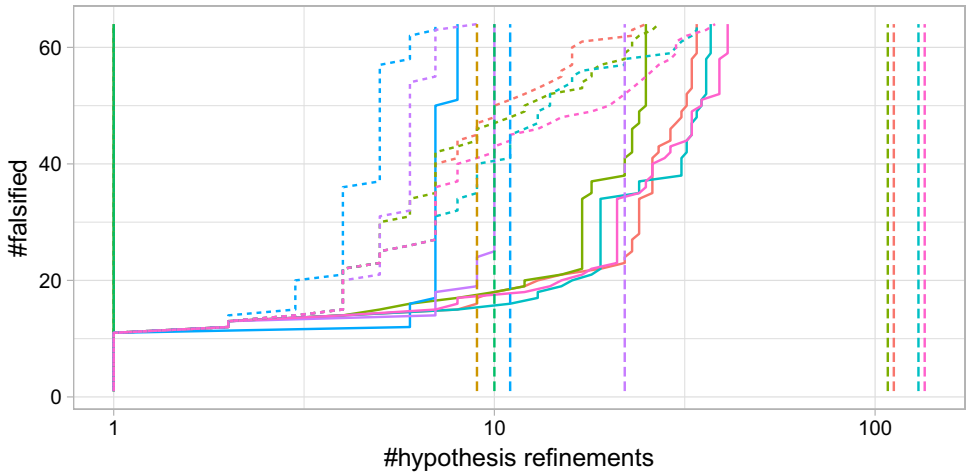


Fig. 12 Number of *hypothesis refinements* for *monitoring* on problem instance 5, comparing various AAL algorithms and BBC strategies (color figure online)

shorter. This is because these are counterexamples to safety properties, where lassos do not need to be validated on the SUL, only finite prefixes. The counterexamples that are produced with Büchi automata are finite prefixes of their lassos. This means the lengths of counterexamples shown on the x-axis are also the shortest prefixes for which the emptiness oracle was able to deduce their lasso would be accepted by the SUL as well. These shortest prefixes are still quite long, they grow even larger on larger RERS problem instances, while the lengths of counterexamples produced with monitors remain quite small. One can see there are 7 properties that could only be disproved with Büchi automata. These 7 properties are liveness properties, not safety properties.

The length of counterexamples also influences the performance of learning algorithms. Figures 14 and 15 show how many learning queries are required to disprove a certain number of properties and answers the fifth research question. In these figures, each learning algorithm has three line types. One where Büchi automata are used, one where monitors are used and one where both monitors and Büchi automata are used. The lines, where only monitors are used, are not as high as those where Büchi automata are used. Indeed,

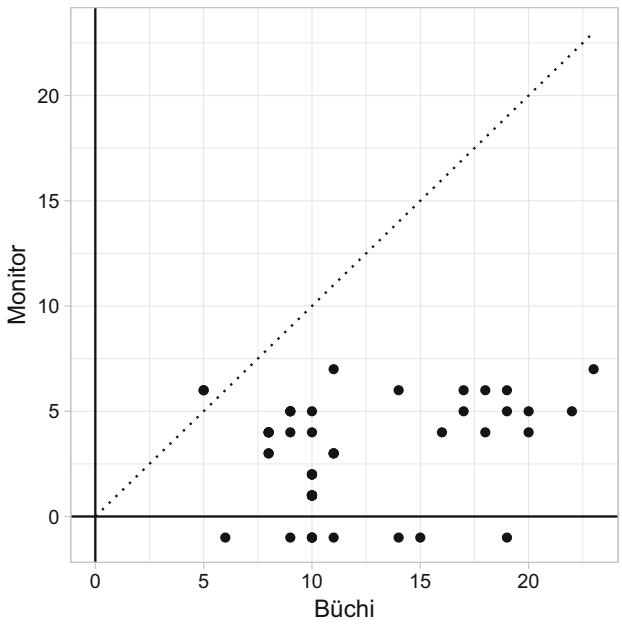


Fig. 13 Length of counterexamples with *ADT* and `DisproveFirstOracle` on problem instance 1, comparing various Automata types (monitors, Büchi automata)

Fig. 14 Legend for model checking and monitoring (color figure online)

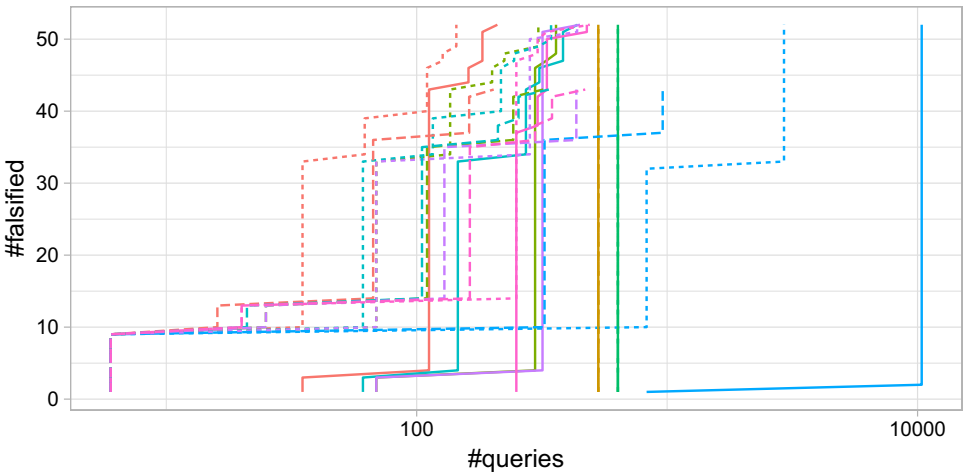


Fig. 15 Number of *learning* queries with `DisproveFirstOracle` on problem instance 1, comparing various Automata types (monitors, Büchi automata, or both) (color figure online)

all those lines are missing exactly those counterexamples that are below the line $y = 0$ in Fig. 13. Further observations that can be made are that often is better to try to disprove properties first with monitors and that the combination of monitors and Büchi automata is better than solely using Büchi automata and monitors.

With all this data we can conclude that it is best to use the *ADT* learning algorithm, the `DisproveFirstOracle` black-box checking strategy and to use monitors. Furthermore, model checking with Büchi automata should be used as a complementary method to further improve the completeness of the method in the presence of liveness properties.

6 Conclusion and future work

We have presented a sound black-box checking method and an implementation with the LearnLib and LTSmin. The bottom line is that black-box checking exploits the knowledge in the given LTL properties to speed up the learning process.

We presented novel sound approaches for liveness LTL properties, where we can check if a system under learning accepts an infinite lasso-shaped word. Our method for liveness contrasts the original proposal where a (hard to guess) upper-bound on the number of states of the system under learning is assumed. Additionally one can now make use of monitors that may provide finite length counterexamples to safety properties. Using monitors further improves the speed of learning algorithms in terms of learning queries. Our implementation is available under a liberal free and open source license, such that it can be put to practice quite easily. Software testers now have a free ease-of-use *sound* black-box checking implementation available for industrial use cases.

Our experimental results reveal three conclusions: (1) the recent learning algorithms ADT, and TTT perform the best in a black-box checking setting. Furthermore, (2) trying to disprove all properties on a hypothesis before refining the hypothesis is better than trying to refine hypotheses when a counterexample is spurious. Finally, (3) The use of monitoring before full LTL model checking leads to a considerable reduction in the number of learning queries.

In contrast to some other learning algorithms in the LearnLib, ADT, and TTT are incremental learning algorithms, meaning they construct more hypotheses while using fewer learning queries. In an active learning setting, this may degrade performance, because more equivalence queries are required. In a black-box checking setting this appeared to be an advantage, because model checking queries replace expensive equivalence queries.

Future work includes performing experiments on a larger class of applications. Furthermore, another application of our work could be to use our notion of ω-queries with loop detection to implement the ω-queries in the theory of ω-regular learning [2,26]. Another direction is more practical; a usability study in an industrial context should be done, to assess how appropriate the extension to the LearnLib is there.

There are still several potential improvements of our method that could be explored. First, experiments could be performed to compare ADT and TTT with the IKL algorithm in LBTest. Similarly, one could try other model checkers such as NuSMV and SPIN, to investigate the effects of their counterexample generation. Another strategy could be to generate *all* counterexamples to a given property, and use many spurious counterexamples to refine the hypothesis. Currently, the black-box checking approach is inherently asymmetric, in the sense that model checking only provides counterexamples that are in the language of the hypothesis but not in the SUL. It would be interesting if it were possible to use the LTL properties for detecting words that are in the SUL but missing in the hypothesis. Finally, in our case study, we applied a perfect state equivalence function to the RERS problems, but it would be interesting to apply our approach to cases where only part of the state can be observed, or when the SUL is hardware, instead of software.

Acknowledgements We would like to thank Markus Frohme for his excellent work on maintaining the LearnLib and the AutomataLib, reviewing our code in pull requests and fine-tuning the LearnLib's API extensions presented in this work.

References

1. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 75(2):87–106
2. Angluin D, Fisman D (2016) Learning regular omega languages. Theor Comput Sci 650:57–72
3. Baier C, Katoen J (2008) Principles of model checking. MIT Press, Cambridge
4. Belinfante A (2014) JTorX: exploring model-based testing. In: PhD thesis, University of Twente, Enschede, Netherlands
5. Bloemen V, van de Pol J (2016) Multi-core scc-based LTL model checking. Haifa Verif Conf Lect Notes Comput Sci 10028:18–33
6. Bollig B, Katoen J, Kern C, Leucker M, Neider D, Piegdon DR (2010) libalf: The automata learning framework. CAV Lect Notes Comput Sci 6174:360–364
7. Cimatti A, Clarke EM, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) Nusmv 2: an opensource tool for symbolic model checking. CAV Lect Notes Comput Sci 2404:359–364
8. Courcoubetis C, Vardi MY, Wolper P, Yannakakis M (1992) Memory-efficient algorithms for the verification of temporal properties. Form Methods Syst Des 1(2/3):275–288
9. Couvreur J (1999) On-the-fly verification of linear temporal logic. Lect Notes Comput Sci World Congr Form Methods 1708:253–271
10. Duret-Lutz A, Lewkowicz A, Fauchille A, Michaud T, Renault E, Xu L (2016) Spot 2.0: a framework for LTL and ω-automata manipulation. ATVA Lect Notes Comput Sci 9938:122–129
11. Frohme M (2015) Active automata learning with adaptive distinguishing sequences. Master's thesis, Technische Universität Dortmund
12. Fujiwara S, von Bochmann G, Khendek F, Amalou M, Ghedamsi A (1991) Test selection based on finite state models. IEEE Trans Softw Eng 17(6):591–603
13. Groce A, Peled DA, Yannakakis M (2006) Adaptive model checking. Log J IGPL 14(5):729–744
14. Holzmann GJ (2004) The SPIN model checker: primer and reference manual. Addison-Wesley, Boston
15. Howar F (2012) Active learning of interface programs. PhD thesis, Dortmund University of Technology, Dortmund
16. Howar F, Steffen B (2018) Active automata learning in practice: an annotated bibliography of the years 2011 to 2016. Lect Notes Comput Sci Mach Learn Dyn Softw Anal 11026:123–148
17. Huistra D, Meijer J, van de Pol J (2018) Adaptive learning for learn-based regression testing. FMICS Lect Notes Comput Sci 11119:162–177
18. Isberner M (2015) Foundations of active automata learning: an algorithmic perspective. In: PhD thesis, Technical University Dortmund, Germany
19. Isberner M, Howar F, Steffen B (2014) The TTT algorithm: a redundancy-free approach to active automata learning. RV Lect Notes Comput Sci 8734:307–322
20. Isberner M, Howar F, Steffen B (2015) The open-source learnLib: a framework for active automata learning. CAV Lect Notes Comput Sci 9206:487–495
21. Jasper M, Fecke M, Steffen B, Schordan M, Meijer J, van de Pol J, Howar F, Siegel SF (2017) The RERS 2017 challenge and workshop (invited paper). In: SPIN, ACM, pp 11–20
22. Kant G, Laarman A, Meijer J, van de Pol J, Blom S, van Dijk T (2015) Ltsmin: High-performance language-independent model checking. TACAS Lect Notes Comput Sci 9035:692–707
23. Kearns MJ, Vazirani UV (1994) An introduction to computational learning theory. MIT Press, Cambridge
24. Khosrowjerdi H, Meinke K, Rasmusson A (2017) Learning-based testing for safety critical automotive applications. IMBSA Lect Notes Comput Sci 10437:197–211
25. Laarman A, Langerak R, van de Pol J, Weber M, Wijs A (2011) Multi-core nested depth-first search. ATVA Lect Notes Comput Sci 6996:321–335
26. Maler O, Pnueli A (1995) On the learnability of infinitary regular sets. Inf Comput 118(2):316–326
27. Meijer J, van de Pol J (2018) Sound black-box checking in the learnlib. NFM Lect Notes Comput Sci 10811:349–366
28. Meinke K (2017) Learning-based testing of cyber-physical systems-of-systems: a platooning study. EPEW Lect Notes Comput Sci 10497:135–151
29. Meinke K, Sindhu MA (2011) Incremental learning-based testing for reactive systems. TAP Lect Notes Comput Sci 6706:134–151
30. Meinke K, Sindhu MA (2013) Lbtest: a learning-based testing tool for reactive systems. In: ICST, IEEE computer society, pp 447–454
31. Merten M, Howar F, Steffen B, Margaria T (2011) Automata learning with on-the-fly direct hypothesis construction. ISoLA Workshops Commun Comput Inf Sci 336:248–260
32. Peled DA, Vardi MY, Yannakakis M (2002) Black box checking. J Autom Lang Comb 7(2):225–246
33. van de Pol J, Meijer J (2019) Synchronous or alternating? LTL black-box checking of Mealy machines by combining the LearnLib and LTSmin, Accepted
34. Raffelt H, Merten M, Steffen B, Margaria T (2009) Dynamic testing via automata learning. STTT 11(4):307–324
35. Rivest RL, Schapire RE (1993) Inference of finite automata using homing sequences. Inf Comput 103(2):299–347
36. Shahbaz M, Groz R (2009) Inferring mealy machines. FM Lect Notes Comput Sci 5850:207–222

37. Sindhu MA (2013) Algorithms and tools for learning-based testing of reactive systems. In: PhD thesis, KTH Royal Institute of Technology, Sweden
38. Steffen B, Howar F, Merten M (2011) Introduction to active automata learning from a practical perspective. SFM Lect Notes Comput Sci 6659:256–296
39. Steffen B, Isberner M, Naujokat S, Margaria T, Geske M (2014) Property-driven benchmark generation: synthesizing programs of realistic structure. STTT 16(5):465–479
40. Steffen B, Jasper M, Meijer J, van de Pol J (2017) Property-preserving generation of tailored benchmark petri nets. In: ACSD, IEEE computer society, pp 1–8
41. Tabakov D, Vardi MY (2010) Optimized temporal monitors for systemc. RV Lect Notes Comput Sci 6418:436–451
42. Tarjan RE (1972) Depth-first search and linear graph algorithms. SIAM J Comput 1(2):146–160
43. Timmer M, Brinksma E, Stoelinga M (2011) Model-based testing. In: Software and systems safety: specification and verification, NATO science for peace and security series-D: information and communication security, vol 30, IOS Press, Amsterdam, pp 1–32