Probabilistic black-box reachability checking (extended version)

Bernhard K. Aichernig · Martin Tappler

Published online: 21 May 2019
© The Author(s) 2019

Abstract
Model checking has a long-standing tradition in software verification. Given a system design, it checks whether desired properties are satisfied. Unlike testing, it cannot be applied in a black-box setting. To overcome this limitation, Peled et al. introduced black-box checking, a combination of testing, model inference and model checking. The technique requires systems to be fully deterministic. For stochastic systems, statistical techniques are available. However, they cannot be applied to systems with non-deterministic choices. We present a black-box checking technique for stochastic systems that allows both, non-deterministic and probabilistic behaviour. It involves model inference, testing and probabilistic model-checking. Here, we consider reachability checking, i.e., we infer near-optimal input-selection strategies for bounded reachability.

Keywords Model inference · Statistical model-checking · Reachability analysis · Black-box checking · Testing · Verification

1 Introduction

Model checking has a long-standing tradition in software verification. Given a system design, model-checking techniques determine whether requirements stated as formal properties are satisfied. These techniques and other forms of model-based verification fall short if no design is available. Model learning provides a solution to this issue. It establishes the basis for model-based verification by automatically learning automata models of black-box systems from observed data. Data used as basis for learning is usually given in the form of system traces, that is, sequences of system events, which may be partitioned into input and output events. Note that model learning is also often referred to as model inference, thus we use both terms interchangeably.

There are two main forms of model learning: passive learning and active learning. Passive learning learns from preexisting data such as system logs, while active learning actively queries the system that is examined to gain relevant information. This can for instance be done via testing. Noteworthy early examples of passive learning techniques are RPNI for...
regular languages \[27,41\] and Alergia [11] for stochastic regular languages, which learn deterministic finite automata (DFAs) and their stochastic counterparts, respectively. Both, RPNI and Alergia, apply a principle called state-merging. More recent work based on this principle extends the applicability of passive model learning to timed systems [52], Moore machines [24] and to stochastic systems involving non-deterministic choices [35,36], which we use in this article. All these approaches have in common that the models they learn depend on given sampled training data.

In contrast to this, active learning approaches rely on the possibility to query relevant information. Angluin formalised this by introducing the minimally adequate teacher framework in her seminal work on the $L^*$ algorithm [4]. This framework assumes the existence of a teacher that is able to answer two types of queries: membership queries and equivalence queries. When a model of a software system is learned, membership queries basically check whether a given traces can be observed and equivalence queries check whether a hypothesised system model is equivalent to the system under investigation. In practice, both queries are usually implemented via testing. Since the introduction of $L^*$, it has been adapted and extended to various types of systems like Mealy machines [37,46], timed systems [25] and non-deterministic systems [28,53]. There are also $L^*$-based learning approaches applicable for probabilistic system models [9,19], but they place strong assumptions on the information that can be queried. These approaches are therefore unsuitable considering a testing scenario, which allows interaction with a black-box system only via testing.

In this paper, we consider such a testing scenario, in which we know the interface of a black-box system and we can gain information by testing the system. Furthermore, we assume that inputs to the system can be freely chosen and that reactions are stochastic. This makes Markov decision processes (MDPs) a well-suited choice of model type. MDPs allow for non-deterministic choices of inputs, while state transitions are stochastic, whereby outputs are produced depending on the entered state. Given such a system, we aim at generating testing strategies that produce desired outputs with high probability. For learning, we rely on an adaptation of Alergia [11] called IOAlergia [35,36], which learns MDPs. While this learning technique is passive in general, our technique is active, as we generate new data for learning by testing. In an iterative approach, we steer the data generation based on learned models towards desired outputs to explore relevant parts of the system more thoroughly. That way, we aim at iteratively improving the accuracy of learning with respect to these outputs. This is in contrast to the application of IOAlergia in an active setting by Chen and Nielsen [13], as they aimed at actively improving the overall accuracy of learned models.

Model learning enables various forms of verification for black-box systems, such as differential equivalence testing on model-level [5,48,49], model-checking [20,21] and model-based testing with learned models [1]. A particularly interesting technique combining model learning, model checking and testing is black-box checking introduced by Peled et al. [42]. This technique learns models of black-box systems in the form of DFAs on-the-fly and iteratively via $L^*$. Whenever a hypothesis automaton model is created, the hypothesis is model checked which may reveal a fault in the system or show that learning was incomplete and needs to be continued. If model checking does not reveal a fault, equivalence between the hypothesis and the black-box system is checked via testing. In case, non-equivalence is detected the learned hypothesis is extended and learning continues.

Inspired by black-box checking, we propose an approach to analyse reactive systems exhibiting stochastic behaviour in a black-box setting. We also follow a learning-based approach involving repeated testing and probabilistic model-checking. Instead of targeting general properties, e.g., formulated in probabilistic temporal logics, we check reachability as a first step. Since we follow a simulation-based approach, we check bounded reachability.
Rather than learning DFAs, we assume that systems can be modelled by MDPs. Hence, we consider systems controllable by inputs, chosen by an environment or a tester. As such, these systems involve non-determinism resulting from the choice of inputs and stochastic behaviour reflected in state transitions and outputs. Given such a system, our goal in bounded reachability checking is to find an input-selection strategy, i.e. a resolution of non-determinism, which maximises the probability of reaching a certain property within a bounded number of steps. Properties can, for example, be the observation of desired outputs. A possible application scenario for our technique is stress testing of systems with stochastic failures. We could generate a testing strategy that provokes such failures.

The approach we follow is shown in Fig. 1. First, we sample system traces randomly. Then, we infer an MDP from these traces via the state-merging-based method described by Mao et al. [35,36], which as noted above is called IOALERGIA. Once we inferred a hypothesis model $M_{h1}$, we use the PRISM model checker [29] for a reachability analysis to find the maximal probability of reaching a state satisfying a property $\psi$. PRISM computes a probability $p$ and a strategy $s_1$ (also called adversary or scheduler) to reach $\psi$ with $p$. Since IOALERGIA infers models from system traces, the quality of the model $M_{h1}$ depends on these traces. If $\psi$ is not adequately covered, $s_1$ inferred from $M_{h1}$ may perform poorly and rarely reach $\psi$. To account for that, we follow an incremental process. After initial random sampling, we iteratively infer models $M_{hi}$ from which we infer strategies $s_i$. To sample new traces for $M_{h_{i+1}}$ we select inputs randomly and based on $s_i$, that is, we use the strategy $s_i$ for directed testing. Selecting inputs with $s_i$ ensures that paths relevant to $\psi$ will be explored more thoroughly. This process is repeated until either a maximal number of rounds $n$ has been executed, or a heuristic detects that the search has converged to a scheduler.

We mainly use PRISM to generate strategies, but ignore the probabilities computed in the reachability analysis. Since the computations are based on possibly inaccurate learned models, the probabilities may significantly differ from the true probabilities. Strategies, however, may serve as testing strategies regardless of the accuracy of the learned models. In fact, we evaluate the final strategy generated in the process described above via directed testing of the system under test (SUT). Since the behaviour under a strategy is purely probabilistic, this is a form of Monte Carlo simulation, which is commonly used in statistical model-checking (SMC) [31]. The evaluation provides an estimation of the probability of reaching $\psi$ with the
actual SUT under strategy \( s_l \), where \( l \) is the last round that has been executed. By directly interacting with the SUT during evaluation, the computed estimation is an approximate lower bound for the optimal probability. In contrast to this, the reachability probabilities computed by PRISM based on the learned model do not enjoy this property.

In summary, we combine techniques from various disciplines to optimise the probability of observing desired outputs within a bounded number of steps.

| Learning     | We rely on IOAlergia for learning MDPs. This algorithm has been developed with verification in mind and evaluated in a model-checking context [35,36]. |
|--------------|---------------------------------------------------------------------------------------------------------------|
| Probabilistic model-checking | We use PRISM [29], a state-of-the-art probabilistic model checker, to generate strategies for bounded reachability based on learned models. |
| Testing      | Directed sampling guided by a strategy is a form of model-based testing with learned models. The sampling algorithm was developed for the presented technique. |
| Statistical model-checking | We evaluate the final strategy on the SUT. As the SUT is a black-box, we cannot apply probabilistic model-checking and instead perform a Monte Carlo simulation to estimate reachability probabilities, like in SMC [31]. |

Parts of this paper have already been published in the proceedings of the 17th International Conference on Runtime Verification [3]. Additional content presented in the current paper covers the heuristic check for detecting convergence, a more thorough evaluation including two new case studies and several further improvements throughout the paper.

The rest of this paper is structured as follows. In Sect. 2, we will discuss related work. Section 3 introduces preliminaries used in Sect. 4 which discusses the proposed approach. We present evaluation results in Sect. 5. Finally, we provide an outlook on future work and conclude in Sect. 6.

2 Related work

As discussed before, black-box checking [42] is closely related. In contrast to our technique, it considers non-stochastic systems, but more general properties. Various follow-up work demonstrates the potential of learning-based verification. Extensions, e.g., take existing models into account [26], focus on the composition of black-box and white-box components [17], or check security properties [47].

Mao et al. [34–36] also inferred probabilistic models with the purpose of model checking. In fact, we apply the model-inference technique for MDPs described by them. Wang et al. [54] apply a variant of Alergia as well and take properties into account during model inference with the goal of probabilistic model-checking. They apply automated property-specific abstraction/refinement to decrease the model-checking runtime. Nouri et al. [39] also combine stochastic learning and abstraction with respect to some property. Their goal is to improve the runtime of SMC. Notably, their approach could also be applied for black-box systems, but does not consider controllability via inputs. Further work on SMC of black-box systems can be found in [45,55].

Although we did not adapt IOAlergia, a passive model-inference technique, we apply it in an active setting. Chen and Nielsen [13] describe active learning of MDPs based on
IOALERGIA. However, they do not aim at model checking, but try to reduce the required number of samples by directing sampling towards uncertainties.

We try to find optimal schedulers for MDPs. This problem has been solved in other simulation-based verification approaches as well, like in SMC. A lightweight approach for finding schedulers in SMC is described in [14,33]. By representing schedulers efficiently, they are able to consider history-dependent schedulers and through “smart sampling” they accomplish finding near-optimal schedulers with low simulation budget. Brázdil et al. [10] presented an approach to unbounded reachability analysis via SMC. The technique is based on delayed Q-learning, a form of reinforcement learning, requiring only limited knowledge of the system (but more than our technique). Another approach using reinforcement learning for strategy inference for reachability objectives has been presented by David et al. [15]. They minimise expected cost while respecting worst-case time bounds.

Learning-based synthesis of control strategies for MDPs has also been studied by Fu and Topcu [23]. They obtain control strategies which are approximately optimal with respect to linear temporal logic (LTL) specifications. They consider transition probabilities to be initially unknown, but in contrast to our setting they assume the MDP structure to be known.

3 Preliminaries

We introduce background material following [22,36], but consider only finite traces, finite paths, and bounded reachability, as we use a simulation-based approach. The restriction to bounded properties is also commonly found in SMC [31], which is also simulation-based and from which we apply concepts. Moreover, SMC of unbounded properties is especially challenging in a black-box setting [32].

Basics. Let \( \Sigma^{\text{in}} \) and \( \Sigma^{\text{out}} \) be sets of input and output symbols. An input/output string \( s \) is an alternating sequence of inputs and outputs, starting with an output, i.e. \( s \in \Sigma^{\text{out}} \times (\Sigma^{\text{in}} \times \Sigma^{\text{out}})^* \). We denote by \( |s| \) the number of input symbols in \( s \) and refer to it also as string/trace length. Given a set \( S \), we denote by \( \text{Dist}(S) \) the set of probability distributions over \( S \), i.e. for all \( \mu \) in \( \text{Dist}(S) \) we have \( \mu : S \rightarrow [0, 1] \) such that \( \sum_{s \in S} \mu(s) = 1 \). We denote the indicator function by \( 1_A \) which returns 1 for \( e \in A \) and 0 otherwise.

In Sect. 4, we apply two pseudo-random functions \( \text{coinFlip} \) and \( \text{randSel} \). These require an initialisation operation which takes a seed-value for a pseudo-random number generator and which returns implementations of both functions. The function \( \text{coinFlip} \) implements a biased coin flip and is defined for \( p \in [0, 1] \) by \( \mathbb{P}(\text{coinFlip}(p) = \top) = p \) and \( \mathbb{P}(\text{coinFlip}(p) = \bot) = 1 - p \). The function \( \text{randSel} \) takes a set as input and returns a single element of the set, whereby the element is chosen according to a uniform distribution, i.e. \( \forall e \in S : \mathbb{P}(\text{randSel}(S) = e) = \frac{1}{|S|} \).

3.1 Markov decision processes

MDPs allow modelling reactive systems with probabilistic responses. An MDP starts in an initial state. During execution, the environment may choose and execute inputs non-deterministically upon which the system reacts according to its current state and its probabilistic transition function. For that, the system changes its state and produces an output.

**Definition 1** (Markov decision process (MDP)) A Markov decision process (MDP) is a tuple \( M = (Q, \Sigma^{\text{in}}, \Sigma^{\text{out}}, q_0, \delta, L) \) where
Q is a finite set of states,

- \( \Sigma_{\text{in}} \) and \( \Sigma_{\text{out}} \) are finite sets of input and output symbols respectively,

- \( q_0 \in Q \) is the initial state,

- \( \delta : Q \times \Sigma_{\text{in}} \rightarrow \text{Dist}(Q) \) is the probabilistic transition function, and

- \( L : Q \rightarrow \Sigma_{\text{out}} \) is the labelling function.

The transition function \( \delta \) must be defined for all \( q \in Q \) and \( i \in \Sigma_{\text{in}} \). We consider only deterministic MDPs, i.e. \( \forall q \in Q, \forall i : \delta(q, i)(q') > 0 \land \delta(q, i)(q'') > 0 \) implies \( q' = q'' \) or \( L(q') \neq L(q'') \). Non-determinism thus results only from the non-deterministic choice of inputs by the environment.

The above definition requires MDPs to be input-enabled, that is, they must not block or reject inputs. Since we assume SUTs to be MDPs, this allows us to execute any input at any point in time.

We generally set \( \Sigma_{\text{out}} = P(\Sigma) \) where \( \Sigma \) is a set of relevant propositions and \( L(q) \) gives the propositions that hold in state \( q \). A finite path \( \rho \) through an MDP is an alternating sequence of states and inputs, i.e. \( \rho = q_0i_1q_1 \cdots i_{n-1}q_{n-1}i_nq_n \in Q \times (\Sigma_{\text{in}} \times Q)^* \). The set of all paths of an MDP \( \mathcal{M} \) is denoted by \( \text{Path}_\mathcal{M} \). A path \( \rho \) corresponds to a trace \( L(\rho) = t \), i.e. an input/output string, with \( t = o_0i_1o_1 \cdots i_{n-1}o_{n-1}i_no_n \) and \( L(q_i) = o_i \). To reason about probabilities of traces, we need a way to resolve non-determinism. To accomplish this, we introduce schedulers which are often also referred to as adversaries or strategies [36].

Schedulers basically choose the next input action (probabilistically) given a history of visited states, i.e. a path.

**Definition 2 (Scheduler)** Given an MDP \( \mathcal{M} = (Q, \Sigma_{\text{in}}, \Sigma_{\text{out}}, q_0, \delta, L) \), a scheduler for \( \mathcal{M} \) is a function \( s : \text{Path}_\mathcal{M} \rightarrow \text{Dist}(\Sigma_{\text{in}}) \).

To define a probability distribution over finite paths, we need another component, a probability distribution \( p_l \in \text{Dist}(\mathbb{N}_0) \) over the length of paths. Given an MDP \( \mathcal{M} = (Q, \Sigma_{\text{in}}, \Sigma_{\text{out}}, q_0, \delta, L) \), a length probability \( p_l \) and a scheduler \( s \) induce a probability distribution \( \mathbb{P}_{\mathcal{M},s} \) on the set of paths \( \text{Path}_\mathcal{M} \), defined by:

\[
\mathbb{P}_{\mathcal{M},s}(q_0i_1q_1 \cdots i_nq_n) = p_l(n) \prod_{j=1}^{n} s(q_0 \cdots i_{j-1}q_{j-1})(i_j) \cdot \delta(q_{j-1}, i_j)(q_j)
\]  \( (1) \)

Probability distributions over finite paths may instead of \( p_l \), e.g., include state-dependent termination probabilities [34]. We take a path-based view because we actively sample from MDPs. As paths correspond to traces, \( \mathbb{P}_{\mathcal{M},s} \) defines a distribution over traces as well and we control the trace length via \( p_l \).

Since we target reachability, we do not need general schedulers, but may restrict ourselves to memoryless deterministic schedulers [30]. A scheduler is memoryless if its choice of inputs depends only on the current state, i.e. it is a function from \( Q \) to \( \text{Dist}(\Sigma_{\text{in}}) \). It is deterministic if for all \( \rho \in \text{Path}_\mathcal{M} \), there is exactly one \( i \in \Sigma_{\text{in}} \) such that \( s(\rho)(i) = 1 \). Otherwise, it is called randomised. Example 1 describes an MDP and a scheduler for a faulty coffee machine.

**Example 1** Figure 2a shows an MDP modelling a faulty coffee machine. Edge labels denote input symbols and corresponding transition probabilities, whereas output labels are placed above states. After insertion of a coin and pressing a button, the coffee machine is supposed to
provide coffee. However, with a probability of 0.1 it may reset itself instead. A deterministic memoryless scheduler \(s\) may provide inputs \(\text{coin}\) and \(\text{but}\) in alternation, i.e. \(s(q_0) = 1(\text{coin}), s(q_1) = 1(\text{but}),\) and \(s(q_2) = 1(\text{coin})\). By setting \(p_l = 1(2)\), all strings must have length 2, such that, e.g., \(P_{M,s}(\rho) = 0.9\) for \(\rho = q_0 \cdot \text{coin} \cdot q_1 \cdot \text{but} \cdot q_2\).

### 3.2 Model inference

We infer MDPs via an adaptation of Alergia, called IOAlergia \([11,35,36]\). The technique takes input-output strings as input and constructs an input output frequency prefix tree acceptor (IOFPTA) representing the strings. An IOFPTA is a tree with edges labelled by inputs and nodes labelled by outputs. Additionally, edges are annotated with frequencies denoting how often a corresponding string was present in the sample. An IOFPTA with normalised frequencies represents a tree-shaped MDP whereby tree nodes correspond to MDP states.

In a second step, the IOFPTA is transformed through iterated state-merging, which potentially introduces cycles. This step compares nodes in the tree and merges them if they show similar output behaviour such that it is likely that they correspond to the same state in the MDP, generating the data. IOAlergia basically views the IOFPTA as an MDP with non-normalised transition probabilities. During the operation of the algorithm, the states of the MDP are partitioned into three sets: red states which have been checked, blue states which are neighbours of red states, and uncoloured states. Initially, the only red states is the root of the IOFPTA. After initialisation, pairs of blue and red states are checked for compatibility and merged if compatible. Otherwise, the blue one is coloured red. This is repeated until all states are coloured. After normalisation of transition probabilities, IOAlergia returns an MDP.

Two states are compatible if they have the same label, their outgoing transitions are compatible and their successors are recursively compatible. Outgoing transitions are compatible, if their empirical probabilities, estimated from the data, are sufficiently close to each other. In other words, we check for all inputs if the estimated probability distribution over outputs conditioned on inputs are sufficiently similar. If they are, we check recursive compatibility of successors reached by all input-output pairs. A parameter \(\epsilon_{\text{Alergia}} \in (0, 2]\) controls the significance level of a statistical test, which determines whether two empirical probabilities are sufficiently close. We represent calls to IOAlergia by \(\text{IOAlergia}(S, \epsilon_{\text{Alergia}}) = M\) where \(S\) is a multiset of input-output strings and \(M\) is a resulting MDP.

Figure 1 shows an IOFPTA for the coffee machine from Example 1, but sampled with a (uniformly) randomised scheduler and a different \(p_l\). Edge labels denote inputs and associated frequencies, while outputs are placed next to nodes. At first, \(s_1\) might be merged with \(s_0\) as their successors are similar. Redirecting the \(\text{but}\) edge from \(s_1\) to \(s_0\) would create the self loop in the initial state.

![Fig. 2](image_url)  
**Fig. 2** Coffee machine example
3.3 Statistical model-checking

We consider step-bounded reachability. The syntax of formulas \( \phi \) is given by: \( \phi = F^{<k}\psi \) with \( \psi = \neg \psi \lor \psi \lor \psi \lor AP, AP \) denoting an atomic proposition, and \( k \in \mathbb{N} \).

The formula \( \phi = F^{<k}\psi \) denotes that \( \psi \) should be satisfied in a state reached in less than \( k \) steps. We define the satisfaction of \( \phi = F^{<k}\psi \) via: a trace \( t = o_0i_1o_1 \cdots i_{n-1}o_{n-1}i_no_n \) satisfies \( \phi \) denoted by \( t \models \phi \) if there is an \( i < k \) such that \( o_i \models \psi \). The evaluation of a trace \( t \) with respect to a formula \( \phi = F^{<k}\psi \) places restrictions on the length of \( t \). In particular, we can only conclude that \( t \models \phi \) if \( t \) does not contain an \( o_i \) with \( o_i \models \psi \) and contains at least \( k - 1 \) steps. In other words, \( t \) must be long enough to determine that it does not satisfy a reachability property. To ascertain that all traces can be evaluated, we set the length probability \( p_l \) accordingly. We set for all traces: \( p_l(j) = 0 \) for \( j < k - 1 \).

The composition of a scheduler \( s \) and an MDP \( M \) behaves entirely probabilistically. In fact, it induces a discrete time Markov chain (DTMC) [22]. Hence, we can apply techniques from SMC without considering non-determinism. Furthermore, we can define the probability of satisfying a property \( \phi \) with an MDP \( M \), and a scheduler \( s \) by \( P_{M,s}(\phi) = P_{M,s}(\{\rho \in Path_M | L(\rho) \models \phi\}) \) for an appropriate \( p_l \). Note that the value \( P_{M,s}(\phi) \) does not depend on the actual \( p_l \) as long as \( p_l \) ensures that traces are long enough to allow reasoning about satisfaction of \( \phi \).

We estimate this probability via SMC [31]. For that, we associate Bernoulli random variables \( B_i \) with success probability \( p \) with simulations of the SUT. A realisation \( b_i \) is 1 if the corresponding sampled trace satisfies \( \phi \) and 0 otherwise. To estimate \( p = P_{M,s}(\phi) \) we apply Monte Carlo simulation. Given \( n \) individual simulations, the estimate \( \hat{p} = \frac{\sum_{i=1}^{n} b_i}{n} \). In order to bound the error of the estimation with a certain degree of confidence, we compute the number of required simulations based on a Chernoff bound [31,40]. This bound guarantees that if \( p \) is the true probability, then the distance between \( \hat{p} \) and \( p \) is greater than or equal to some \( \epsilon \) with a probability of at most \( \delta \), i.e. \( \mathbb{P}(|\hat{p} - p| \geq \epsilon) \leq \delta \). The required number of simulations \( n \) and the parameters \( \epsilon \) and \( \delta \) are related by \( \delta = 2e^{-2n\epsilon^2} \) [40], i.e. we compute \( n \) by

\[
N = \left[ \frac{\ln(2) - \ln(\delta)}{2\epsilon^2} \right].
\]  

4 Probabilistic black-box reachability checking

We begin with an overview of the approach, discussing each of the steps briefly. Subsequently, we provide in-depth descriptions. For the remainder of this paper, we assume that we interact with an MDP \( M = \langle Q, \Sigma^\text{in}, \Sigma^\text{out}, q_0, \delta, L \rangle \) representing the SUT of which we only know \( \Sigma^\text{in} \). Basically, we try to find an optimal scheduler for satisfying a given reachability property \( \phi = F^{<k}\psi \) with the SUT \( M \). For this purpose, we iteratively sample system traces, learn models from the traces and infer schedulers from the learned models. The inferred schedulers also serve to refine the sampling strategy. This process is also shown in Fig. 1.

1. Create initial samples The step collects a multiset of system traces through interaction with \( M \) by uniformly choosing and executing inputs from \( \Sigma^\text{in} \).
2. For at most $maxRounds$ rounds do

2.1. Infer model

Given the system traces sampled so far, we use IOALERGIA to infer an MDP $M_{hi} = (\bar{Q}_h, \Sigma^{in}, \Sigma^{out}_h, q_{0h}, \delta_h, L_h)$, where $h$ stands for hypothesis and $i \in [1..maxRounds]$ denotes the current round.

2.2. Reachability analysis

Reachability analysis on $M_{hi}$ with PRISM [29]: i.e. we compute the maximum probability $P_{M_{hi}, s_i}(\phi)$ of satisfying $\phi$ and generate the corresponding scheduler $s_i$.

2.3. Sample with scheduler

We extend the multiset of system traces through property-directed sampling. For that, we choose some inputs with scheduler $s_i$ and some randomly. With increasing $i$, we decrease the portion of random choices.

2.4. Check early stop

We may stop before executing $maxRounds$ rounds if a stopping criterion is satisfied. This criterion is realised with a heuristic check for convergence. In this check, we basically determine whether several consecutive schedulers behave similarly.

3. Evaluate

In a last step, we evaluate the most recent scheduler we have generated. For this evaluation, we sample system traces again, but avoid choosing inputs randomly. The relative frequency of satisfying $\phi$ now gives us an estimate for the success probability of executing $M$, the black-box SUT, controlled by scheduler $s_i$, where $i$ is the last round we executed. A Chernoff bound [31,40], which is commonly used in SMC, specifies the required number of samples.

Create initial samples

In the first step, we sample system traces by choosing input actions randomly according to a uniform distribution. Hence, we sample with a scheduler $s_{\text{unif}}$ defined as follows: $\forall q \in \bar{Q}, s_{\text{unif}} : q \mapsto \mu_{\text{unif}}(\Sigma^{in})$ where $\forall i \in \Sigma^{in} : \mu_{\text{unif}} : i \mapsto \frac{1}{\#\Sigma^{in}}$. Sampling is further controlled by the length probability $p_l$ and by the batch size $n_{\text{batch}}$, i.e. the number of traces collected at once. These parameters also affect subsequent sampling. Additionally, we set a seed-value for the initialisation of pseudo-random functions.

As discussed in Sect. 3, we set $p_l(j) = 0$ for $j < k - 1$ if $k$ is the step bound of the property we test for. This would not be necessary for learning but we generally apply this constraint.

The length of suffixes, i.e. the trace extensions beyond $k$, follows a geometric distribution parameterised by $p_{\text{quit}} \in [0, 1]$. Before each step, we stop with probability $p_{\text{quit}}$. Hence, the number of input steps $|t|$ in a trace $t$ is distributed according to $p_l(|t|) = (1 - p_{\text{quit}})^{|t|-k+1} p_{\text{quit}}$ for $|t| \geq k - 1$ and $p_l(|t|) = 0$ otherwise. Both $p_{\text{quit}}$ and $n_{\text{batch}}$ must be supplied by the user. In the following, $S_i$ denotes the multiset of traces created by the $i$th sampling step, and $S_{\text{all}}$ refers to the multiset of all traces. Hence, $S_{\text{all}}$ is initially set to $S_1$, containing $n_{\text{batch}}$ traces distributed according to $\mathbb{P}^{\text{M}, s_{\text{unif}}}$, collected by random testing.

Infer model

In this step, we use IOALERGIA [35,36] to infer an MDP $M_{hi} = (\bar{Q}_h, \Sigma^{in}, \Sigma^{out}_h, q_{0h}, \delta_h, L_h)$, from $S_{\text{all}} = \bigcup_{j \leq i} S_j$, i.e. an approximate system model. Strictly speaking, we infer an MDP with a partial transition function, which we make input-complete with a function complete. The transition function of an inferred MDP may be undefined for some state-input pair if there is no corresponding execution in $S_{\text{all}}$. For this reason, we add transitions to a special state labelled with dontKnow for undefined state-input
pairs. Once we enter that state, we cannot leave it. The label \( \text{dontKnow} \) is more generally a special output label, which is not part of the original output alphabet.

Formally, \( M_1^h = (Q_1^h, \Sigma_{in}, \Sigma_{out}', q_0, \delta, L_1^h) = \text{IOALERGIA}(S_{all}, \epsilon_{ALERGIA}) \) and \( M_{hi} = \text{complete}(M_1^h) \) where \( Q_1^h = Q_1' \cup \{q_{undef}\}, \Sigma_{out} = \Sigma_{out}' \cup \{\text{dontKnow}\} \), with \( \text{dontKnow} \notin \Sigma_{out}' \), \( q_0' = q_0 \), \( \delta_1 = \delta_1' \cup \{(q_{undef}, i) \mapsto 1_{(q_{undef})}|i \in \Sigma_{in}\} \cup \{(q, i) \mapsto 1_{(q_{undef})}q \in Q_1', i \in \Sigma_{in}\), \( \exists d : (q, i) \mapsto d \in \delta_1'\) and \( L_1 = L_1' \cup \{q_{undef} \mapsto \text{dontKnow}\} \).

Following the terminology of active automata learning [4], we refer to \( M_{hi} \) as the current hypothesis. Input completion via \( \text{complete} \) is required by Definition 1, but does not affect the reachability analysis. We aim at maximising the probability of desired events, therefore generated schedulers will not choose to execute inputs leading to the state \( q_{undef} \). This is due to the fact that once we reached \( q_{undef} \), we have a probability of zero to observe anything other than \( \text{dontKnow} \) according to our hypothesis.

**Reachability analysis** Given the current hypothesis inferred in the last step, our implementation of the approach uses the PRISM model checker [29] to derive a scheduler for satisfying the property \( \phi \). This is achieved by performing the following steps in a fully automated manner:

1. Translate \( M_{hi} \) into the PRISM modelling language, whereby we encode
   1.1 states using integers,
   1.2 inputs using commands labelled with actions, and
   1.3 outputs using labels.

2. Since PRISM only supports scheduler generation for unbounded reachability properties, we preprocess the translated \( M_{hi} \) further [10] and encode \( \phi \) as unbounded property:

   2.1 We add a step-counter variable \( \text{steps} \) ranging between 0 and \( k \), where \( k \) is the step bound of the examined property.
   2.2 The variable \( \text{steps} \) is incremented with every execution of an input until the maximal value \( k \) is reached. Once \( \text{steps} = k \), \( \text{steps} \) is left unchanged.
   2.3 We change \( \phi \) to \( \phi' = F(\psi \land \text{steps} < k) \), i.e. we move the bound from the temporal operator to the property that should be reached.

3. Finally, we use the sparse engine of PRISM to compute the maximum probability \( \max_{s} P_{M_{hi}, s}(\phi') \) for satisfying \( \phi' \) and export the corresponding scheduler \( s_{hi} \), i.e. we verify the property \( \text{Pmax}=?[F(\psi \land \text{steps} < k)] \).

These steps implicitly create an MDP \( \mathcal{M}_{steps} \) containing \( k + 1 \) copies \((q, st)\) of each state \( q \) of \( M_{hi} \), one for each value \( st \) the variable \( \text{steps} \) can take. Note that not all \( k + 1 \) copies of a state are reachable. If \( q' \) is reachable from \( q \) in \( M_{hi} \), then \( \text{(q', st+1)} \) is reachable from \( (q, st) \) if \( st < k \). If \( st = k \), then \( (q', st) \) is reachable from \( (q, st) \). The target states in \( \mathcal{M}_{steps} \) for the unbounded reachability property \( \phi' = F(\psi \land \text{steps} < k) \) are all \((q, st)\) with \( L(q) \models \psi \) and \( st < k \). Furthermore, all \((q, st)\) with \( st = k \) are non-target states from which we cannot reach target states, as required by the original bounded reachability property \( \phi \). Given \( \mathcal{M}_{steps} \) and the unbounded reachability property \( \phi' \), PRISM exports memoryless deterministic schedulers. These schedulers, however, do not define input choices for all states, but only for states reachable by the composition of scheduler and corresponding model. To account for cases with undefined scheduler behaviour, we use the notation \( s_{hi}(q) = \bot \). It denotes that scheduler \( s_{hi} \) does not define a choice for \( q \).
Algorithm 1 Property-Directed Sampling

Input: $p_{\text{rand}}$, $M_{hi}$, $S_{hi}$, $n_{\text{batch}}$, $\phi = F \land \psi$, $S_{\text{all}}$, $c_{\text{change}}$

Output: $p_{\text{rand}i+1}$, $M_{hi}$, $S_{\text{all}}$

1: $S_{i+1} \leftarrow \{\}$
2: while $|S_{i+1}| < n_{\text{batch}}$ do
3:     $S_{i+1} \leftarrow S_{i+1} \cup \{\text{SAMPLE}(p_{\text{rand}i}, M_{hi}, S_{hi}, k)\}$
4:     $p_{\text{rand}i+1} \leftarrow p_{\text{rand}i} \cdot c_{\text{change}}$
5:     $S_{\text{all}} \leftarrow S_{\text{all}} \cup S_{i+1}$
6: function SAMPLE($p_{\text{rand}i}, M_{hi}, S_{hi}, k$)
7:     trace $\leftarrow \text{reset}()$
8:     $q_{\text{curr}} \leftarrow q_{\text{hi}}$
9:     while $|\text{trace}| - 1 < k \lor \neg \text{coinFlip}(p_{\text{quit}})$ do
10:        if coinFlip($p_{\text{rand}i}$) $\lor q_{\text{curr}} = \bot \lor S_{hi}(q_{\text{curr}}) = \bot$ then
11:            input $\leftarrow \text{randSel}()$
12:        else
13:            input $\leftarrow S_{hi}(q_{\text{curr}})$
14:        end if
15:        outSut $\leftarrow \text{exec}($input$)$
16:        distQ_{\text{curr}} = S_{hi}(q_{\text{curr}}, \text{input})$
17:        $q_{\text{curr}} \leftarrow \begin{cases} q \in Q_{hi} & \text{such that } L_{hi}(q) = \text{outSut} \land \text{distQ}_{\text{curr}}(q) > 0 \\ \bot & \text{if there is no such } q \end{cases}$
18: end while
19: return trace

Sample with scheduler. Property-directed sampling with inferred schedulers aims at exploring parts of the system more thoroughly that have been identified as being relevant to the property. To avoid getting trapped in local minima, we also explore new paths by choosing actions randomly with probability $p_{\text{rand}i}$, where $i$ corresponds to the current round. This probability is decreased in each round to explore more broadly in the beginning and focus on relevant parts in later rounds. Two parameters control $p_{\text{rand}i}$: $p_{\text{start}} \in [0, 1]$ for the initial probability and $c_{\text{change}} \in [0, 1]$ specifying an exponential decrease, i.e. $p_{\text{rand}1} = p_{\text{start}}$ and $p_{\text{rand}i+1} = c_{\text{change}} \cdot p_{\text{rand}i}$ for $i \geq 1$.

Basically, we execute both SUT and $M_{hi}$ in parallel. The former ensures that we sample traces of the actual system while the latter is necessary because the inferred scheduler $S_{hi}$ is defined for $M_{hi}$. Stated differently, we need to simulate the path taken by the SUT on the current hypothesis $M_{hi}$. This enables selecting actions with $S_{hi}$. As $M_{hi}$ is an approximation, two scenarios may occur in which we cannot use $S_{hi}$. In the following scenarios, we default to choose inputs randomly:

1. The SUT may show outputs not foreseen by $M_{hi}$, i.e. not only probabilities differ. In such cases, we cannot determine the correct state transition in $M_{hi}$.
2. By performing random inputs we may follow a path that is not optimal with respect to $M_{hi}$ and $\phi$. Thus, we may enter a state where $S_{hi}$ is undefined.

The sampling is detailed in Algorithm 1. In addition to artefacts generated by other steps, such as the input-enabled hypothesis $M_{hi}$ and the generated scheduler $S_{hi}$, sampling requires two auxiliary operations:

- reset: resets the SUT to the initial state and returns the unique initial output symbol
- exec: executes a single input changing the state of the SUT and returning the corresponding output

1 If we reach the state labelled $\text{dontKnow}$ during sampling, the outputs of the hypothesis and the SUT are guaranteed to differ. As a result, we will continue sampling with random inputs after reaching that state.
Both operations are realised by a test adapter. Lines 1 to 5 of the algorithm collect traces by calling the \texttt{Sample} function and update $p_{\text{rand}}$. The \texttt{Sample} function returns a single trace, which is created on-the-fly and initialised with the output produced upon resetting the SUT (line 7). Line 8 initialises the current model state. Afterwards, an input is chosen (lines 10 to 13). It is chosen randomly with probability $p_{\text{rand}}$ or if we cannot determine an input (line 10 and 11). Otherwise, the input is selected with $s_{hi}$ (Line 13). We record the output of the SUT in response to the input and extend the trace in lines 14 and 15. The next two lines update the model state. In case, the SUT produces an output which is not allowed by the model, the new model state is undefined (second case in Line 17). This corresponds to the first scenario, in which we default to choose inputs randomly. Trace creation stops if the trace is long enough and if a probabilistic Boolean choice returns \texttt{true} (Line 9), i.e. the actual length follows a probability distribution. Note that lines 10 to 17 implement a randomised scheduler derived from $s_{hi}$. We will refer to this scheduler as \texttt{randomised($s_{hi}$)}. In the taxonomy of Utting et al. [51], property-directed sampling can be categorised as model-checking-based online testing with a combination of requirements coverage and random input-selection as criterion.

\textbf{Evaluate.} As a result of the reachability analysis, PRISM calculates a probability of reaching $\phi$. This probability, however, is derived from a learned model which is possibly inaccurate. Therefore, it may greatly differ from the actual probability of reachability with the SUT. To account for that, we evaluate the scheduler $s_{h} = s_{hi}$, where $i$ is the last round we executed. We accomplish this by sampling a multiset of traces $S_{\text{eval}}$, while generally selecting inputs with $s_{hi}$, i.e. we execute Algorithm 1 with $p_{\text{rand}} = 0$. Thereby, we implicitly sample traces from the DTMC induced by the composition of $M$ and \texttt{randomised($s_{hi}$)}. Since this DTMC behaves entirely probabilistic, we can apply SMC. Hence, we estimate $P_{M,\text{randomised($s_{hi}$)}}(\phi)$ by $\hat{p}_{M, s_{h}} = \frac{|\{s \in S_{\text{eval}} | s \models \phi\}|}{|S_{\text{eval}}|}$. To achieve a given error bound $\epsilon_{\text{eval}}$ with a given confidence $1 - \delta_{\text{eval}}$, we compute the required number of samples $|S_{\text{eval}}| = n_{\text{batch}}$ based on a Chernoff bound [40], i.e. we apply (2). The estimation provides an approximate lower bound of the maximal reachability probability with the SUT. We consider $\hat{p}_{M, s_{h}}$ an approximate lower bound, because we know with confidence $1 - \delta_{\text{eval}}$ that $\max_s P_{M, s}(\phi)$ is at least as large as $\hat{p}_{M, s_{h}} - \epsilon_{\text{eval}}$.

\textbf{Check early stop} We have observed that the performance of schedulers usually increases with the total amount of available data. Probability estimations derived with intermediate schedulers showed that schedulers generated in later rounds tend to perform better than those generated in earlier. However, we have also seen fluctuations in these estimations over time, i.e. some schedulers may perform worse than schedulers generated in previous rounds. With increasing number of rounds these fluctuations generally diminish and the estimations converge. Intuitively, this can be explained by the influence of $p_{\text{rand}}$ in Algorithm 1, which controls the probability of selecting random inputs and decreases over time. As this probabilities $p_{\text{rand}}$ approaches zero, we will almost always select inputs with generated schedulers. This will generally only increase the confidence in parts of the system we have already explored, but will not explore new parts and therefore new schedulers are likely to show similar behaviour to previous ones.

Based on these observations, we developed a heuristic check for convergence. If it detects convergence, we stop the iteration early before executing $\texttt{maxRounds}$ rounds. Two simpler checks actually form the basis of the heuristic. The first, called \texttt{SIMILARCHED}, basically compares the scheduler generated in the current round to the scheduler from the previous round and returns \texttt{true} if both behave similarly. The second check, called \texttt{CONV} builds upon
the first and reports convergence if we detect statistically similar behaviour via \textsc{similarSched} in multiple consecutive rounds. The rationale behind this is that schedulers should behave alike after convergence. We check for similarity rather than for equivalence because there may be several optimal inputs in a state and slight variations in transition probabilities in the inferred models may lead to the different choices of inputs. Furthermore, we can compare schedulers during sampling by comparing whether they would choose the same inputs. This gives us a large number of events as basis for our decision and does not require additional sampling.

The convergence check has three parameters: \( \alpha_{\text{conv}} \) controlling the confidence level, an error bound \( \epsilon_{\text{conv}} \), and a bound on the number of rounds \( r_{\text{conv}} \). The first two parameters control a statistical test which checks whether two schedulers behave similarly. For this test, we consider Bernoulli random variables \( E_i \) for \( i \in [2..\text{maxRounds}] \). \( E_i \) is equal to one if two consecutive schedulers \( s_{hi} \) and \( s_{hi-1} \) behave equivalently, i.e. choose the same input in some state, and zero otherwise. Let \( p_{E_i} \) be the success probability, the probability of \( E_i \) being equal to one. We observe samples of \( E_i \) in Line 13 of Algorithm 1. Each time we choose an input \( i \) with \( s_{hi} \), we also determine which input \( i' \) the previous scheduler \( s_{hi-1} \) would have chosen. We record a positive outcome if \( i = i' \) and a negative outcome otherwise.

Let \( \hat{p}_{E_i} \) be the relative number of positive outcomes, which is an estimate of \( p_{E_i} \). If \( p_{E_i} \) is equal to one, then both schedulers behave equivalently, they always choose the same input. Consequently, we test whether \( \hat{p}_{E_i} \) is close to one. We test the null hypothesis \( H_0 : p \leq 1 - \epsilon_{\text{conv}} \) against \( H_1 : p > 1 - \epsilon_{\text{conv}} \) with a confidence level of \( 1 - \alpha_{\text{conv}} \). The hypothesis \( H_1 \) denotes that the compared schedulers choose the same inputs in most of the cases. Let \( \text{SIMILARSCHED}(\alpha_{\text{conv}}, \epsilon_{\text{conv}}, i) \) be the result of this test in round \( i \), which is \textit{true} if \( H_0 \) is rejected and \textit{false} otherwise.

Finally, we can formulate the complete convergence check \( \text{CONV}(\alpha_{\text{conv}}, \epsilon_{\text{conv}}, i) \). It returns \textit{true} in round \( i \) if \( r_{\text{conv}} \) consecutive calls of \textsc{SIMILARSCHED} returned \textit{true}, i.e.

\[ \text{CONV}(\alpha_{\text{conv}}, \epsilon_{\text{conv}}, i) = \bigwedge_{j=i-r_{\text{conf}}+1}^{i} \text{SIMILARSCHED}(\alpha_{\text{conv}}, \epsilon_{\text{conv}}, j). \]

Note that \( p_{\text{rand}_i} \) implicitly affects the convergence check. We collect samples of \( E_i \) in Line 13 of Algorithm 1, thus large \( p_{\text{rand}_i} \) cause Line 13 to be executed infrequently. As a result, sample sizes of \( E_i \) are small. This influence on the convergence check is beneficial because schedulers are more likely to improve if \( p_{\text{rand}_i} \) is large, as new parts of the system may be explored via frequent random steps.

While the check introduces further parameters, it may simplify the application of the approach in scenarios where we have little knowledge about the system at hand. In such cases, it may be difficult to find a reasonable choice for the number of rounds \( \text{maxRounds} \). With this heuristic, it is possible to choose \( \text{maxRounds} \) conservatively, but stop early once convergence is detected. However, it may also impair results, if convergence is detected too early.

**Convergence to the true model** Generally, Mao et al. [36] showed convergence in the large sample limit for IOAlergia. However, the sampling mechanism needs to ensure that sufficiently many executions of all inputs in all states are observed. This is also discussed in [35], which states that IOAlergia requires a \textit{fair} schedulers, one that chooses each input infinitely often. The uniformly randomised \( s_{\text{unif}} \) satisfies this requirement. As a result, we have convergence in the limit, if we perform only a single round of inference, in which we sample with \( s_{\text{unif}} \).

Property-directed sampling favours certain inputs with increasing number of rounds, but it also selects random inputs with probability \( p_{\text{rand}_i} \) in round \( i \). If we ensure that \( p_{\text{rand}_i} \) is always non-zero, we will select all inputs infinitely often in an infinite number of rounds. Therefore,
the inferred models will converge to the true model (up to bisimulation equivalence) and the inferred schedulers will converge to the optimal scheduler.

Another way to approach convergence is to follow a hybrid approach, by collecting traces via property-directed sampling and via uniform sampling in parallel. Uniform sampling ensures that all inputs are executed sufficiently often, which entails convergence. Property-directed sampling explores parts of the system, identified to be relevant, which increases the confidence in the correct inference of those parts. As a result, intermediate schedulers are more likely to perform well.

Hence, we have convergence in the limit under certain assumptions. In practice, i.e. when learning from limited data, uniform schedulers are likely to be insufficient, if events occur only after long interaction scenarios. If events occur rarely in the sampled system traces, then it is unlikely that the part modelling those events is accurately learned. Active learning, as described by Chen and Nielsen [13], addressed this issue by guiding sampling so as to reduce the uncertainty in the learned model. Our approach similarly guides sampling, but with the aim at reducing uncertainty along traces, which are likely to satisfy a reachability property.

As noted above, we have seen that the inferred schedulers usually converge to a scheduler, which may not be globally optimal, though. We also performed experiments with the outlined hybrid approach to avoid getting trapped in local maxima, by collecting half the system traces through uniform sampling. While it showed favourable performance in a few cases, the incremental approach generally produced better results with the same number of samples. Therefore, we will not discuss experiments with the hybrid approach.

Apart from convergence, it may not always be necessary to find a (near-) optimal scheduler. A requirement may state that the probability of reaching an erroneous state must be smaller than some $p$. Once we found and evaluated a scheduler $s_h$ such that the estimation $\hat{p}_{M,s_h} \geq p$, we basically show with some confidence that the requirement is violated. Such a requirement could be the basis of another stopping criterion. If in round $i$, a sufficiently large number of the sampled traces $S_i$ reaches an erroneous state, we may decide to evaluate the corresponding scheduler $s_{h_{i-1}}$. We could then stop if $\hat{p}_{M,s_{h_{i-1}}} \geq p$ and continue otherwise.

**Application and choice of parameters** We will now briefly discuss the choice of parameters taking our findings into account. A summary of all parameters along with a concise description is given by Table 1.

The product $n_s = n_{batch} \cdot maxRounds$ defines the overall maximum number of samples for inference, thus it could be chosen as large as the testing/simulation budget permits. Increasing $maxRounds$ while fixing $n_s$ increases the time required for learning and model checking. Intuitively, it improves accuracy as well, as sampling is more frequently adjusted towards the considered property. For the systems examined in Sect. 5, values in the range between 50 and 200 led to reasonable accuracy while incurring acceptable runtime overhead. Runtime overhead is the time spent learning and model checking, as opposed to the time spent doing actual testing, i.e. (property-directed) sampling. The convergence check takes three parameters as input for which we identified well-suited default parameters. To ensure high confidence for the statistical test, we set $\alpha_{conv} = 0.01$. Since schedulers should choose the same input in most cases, $\epsilon_{conv}$ should be small, but greater than zero to allow for some variation. In our experiments, we set it to $\epsilon_{conv} = 0.01$ and we set $r_{conv} = 6$. More conservative choices would be possible at the expense of performing additional rounds.

The value of $p_{start}$ should generally be larger than 0.5, while $c_{change}$ should be close to 1. This ensures broad exploration in the beginning and more directed exploration afterwards. Finally, the choice of $p_{quit}$ depends on the simulation budget and the number of inputs. If there is a large number of inputs, it may be highly improbable to reach certain states within a
Table 1 All parameters with short descriptions

| Parameter | Description |
|-----------|-------------|
| \(n\)_{batch} | Number of traces sampled in one round |
| \(maxRounds\) | Maximum number of rounds |
| \(p_{start}\) | Initial probability of random input selection |
| \(c_{change}\) | Factor changing the probability of random input selection |
| \(p_{quit}\) | Parameter of geometric distribution of sampled trace length |
| \(\epsilon_{ALERGIA}\) | Controls significance level of statistical compatibility check of IOALERGIA |
| \(1 - \alpha_{conv}\) | Confidence level of convergence check |
| \(\epsilon_{conv}\) | Error bound of convergence check |
| \(1 - \delta_{eval}\) | Confidence level of scheduler evaluation (Chernoff bound) |
| \(\epsilon_{eval}\) | Error bound of scheduler evaluation (Chernoff bound) |
| \(r_{conv}\) | Number of rounds considered in convergence check |

small number test steps via random testing. Consequently, we should allow for the execution of long tests, in order to reach states requiring complex combinations of inputs. Domain knowledge may also aid in choosing this parameter. If we, e.g., expect a long initialisation phase, \(p_{quit}\) should be low to ascertain that we reach states following the initialisation.

5 Experiments

We evaluated our approach on five case studies from the area of automata learning, control policy synthesis, and probabilistic model-checking. For the first case study, we created our own model of the slot machine described by Mao et al. [36] in the context of learning MDPs. Two case studies consider models of network protocols enhanced with stochastic failures. For that, we transformed deterministic Mealy-machine models as detailed below. The model used in the fourth case study is inspired by the gridworld example, for which Fu and Topcu synthesised control strategies [23]. Finally, we generate schedulers for a consensus protocol [6] which serves as a benchmark in probabilistic model-checking. We discussed the experiments involving the slot machine and the network protocol models before [3]. New additions in this extended version are experiments with the gridworld example, the consensus protocol, and experiments with the convergence check. Note that due to changes of the implementation, measurement result may differ from those in [3].

Adding stochastic failures Deterministic Mealy-machines serve as the basis for two case studies. These Mealy machines are results from previous learning experiments [20,49] and model communication protocols. Basically, we simulate stochastic failures by adding outputs represented by the label \(\text{crash}\). These occur with a predefined probability instead of the correct output. Upon such a failure, the system is reset. We implemented this by transforming the Mealy machines as follows:

1. Translate Mealy machine into Moore machine: this effectively creates an MDP \(\mathcal{M} = (Q, \Sigma^{\text{in}}, \Sigma^{\text{out}}, q_0, \delta, L)\) with a non-probabilistic \(\delta\).
2. Extend \(\Sigma^{\text{out}}\) with a new symbol \(\text{crash}\) and add \(q_{cr}\) to \(Q\) with \(L(q_{cr}) = \text{crash}\).
3. For a predefined probability \(p_{cr}\) and for all \(o\) in a predefined set \(\text{Crashes}\):
3.1. Find all $q, q' \in Q, i \in \Sigma^\text{in}$ such that $\delta(q, i)(q') = 1$ and $L(q') = 0$.

3.2. Set $\delta(q, i)(q') = 1 - p_{\text{cr}}$ and $\delta(q, i)(q_{\text{cr}}) = p_{\text{cr}}$.

3.3. For all $i \in \Sigma^\text{in}$ set $\delta(q_{\text{cr}}, i)(q_{\text{cr}}) = p_{\text{cr}}$ and $\delta(q_{\text{cr}}, i)(q_0) = 1 - p_{\text{cr}}$.

This simulates stochastic failures of outputs belonging to a set $\text{Crashes}$. Instead of producing the correct outputs, we output crash and reach state $q_{\text{cr}}$ with a certain probability. With the same probability we stay in this state and otherwise we reset the system to $q_0$ after the crash.

**Measurement setup and criteria** We have complete information about all models. This allows us to compare our results to optimal values. Nevertheless, for the evaluation we treat the systems as black boxes. The state spaces of the models without step-counter variables for bounded reachability are of sizes 471 (slot machine), 63 (MQTT), 157 (TCP), 35 (gridworld), and 272 (consensus protocol), respectively. For each of these systems, we identified an output relevant to the application domain and applied the presented technique to reach states emitting this output with varying numbers of steps. The slot machine grants prizes and we generated strategies to observe the rarest prize. Using the steps discussed above, we seeded stochastic failures into the MQTT and TCP models, which we tried to reach. The gridworld we used in the evaluation contains a dedicated goal location that served as a as reachability objective. In case of the consensus protocol, we generated strategies to finish the protocol, i.e. reach consensus, with high probability.

For a black-box MDP $\mathcal{M}$ and a reachability property $\phi$, we compare four approaches to find schedulers $s$ for $\mathbb{P}_{\mathcal{M},s}(\phi)$:

- **Incremental Scheduler Inference** We apply the incremental approach discussed in Sect. 4 with a fixed number of rounds. Inferred schedulers are denoted by $s_{\text{inc}}$.

- **Incremental with Convergence Check** We apply the incremental approach, but stop if we either detect convergence with CONV or if $\text{maxRounds}$ rounds have been executed. Inferred schedulers are denoted by $s_{\text{conv}}$.

- **Monolithic Scheduler Inference** To check if the incremental refinement of inferred models pays off, we use the same approach but set $\text{maxRounds} = 1$. In other words, we sample traces by solely choosing inputs randomly. Based on this, we perform a single round, inferring a model and a scheduler which we evaluate. To balance the simulation budget, we collect $\text{maxRounds} \cdot n_{\text{batch}}$ traces, where $\text{maxRounds}$ and $n_{\text{batch}}$ are the parameter settings for inferring $s_{\text{inc}}$. We denote monolithically inferred schedulers by $s_{\text{mono}}$.

- **Uniform Schedulers** As a baseline for comparison we compare to the randomised scheduler $s_{\text{unif}}$ which chooses inputs according to a uniform distribution. This resembles random testing without additional knowledge.

Furthermore, let $s_{\text{opt}} = \arg\max_s \mathbb{P}_{\mathcal{M},s}(\phi)$ be the optimal scheduler. This is the optimal scheduler obtained for the generating model $\mathcal{M}$. As the most important measure of quality, we compare estimates of $\mathbb{P}_{\mathcal{M},s}(\phi)$ to the maximal probability $\mathbb{P}_{\mathcal{M},s_{\text{opt}}}(\phi)$. We consider
Table 2  General parameter settings for experiments

| Parameter   | Value |
|-------------|-------|
| $p_{\text{start}}$ | 0.75  |
| $c_{\text{change}}$ | 0.95  |
| $\epsilon_{\text{Alergia}}$ | 0.5   |
| $\alpha_{\text{conv}}$ | 0.01  |
| $\epsilon_{\text{conv}}$ | 0.01  |
| $\delta_{\text{eval}}$ | 0.01  |
| $\epsilon_{\text{eval}}$ | 0.01  |
| $r_{\text{conv}}$ | 6     |

a scheduler $s$ to be near optimal, if the estimate $\hat{P}_{\mathcal{M},s}$ of $P_{\mathcal{M},s}(\phi)$ derived via SMC is approximately equal to $P_{\mathcal{M},s_{\text{opt}}}(\phi)$, i.e. $|\hat{P}_{\mathcal{M},s} - P_{\mathcal{M},s_{\text{opt}}}(\phi)| \leq \epsilon$, for an $\epsilon > 0$. In the following, we use $\epsilon = \epsilon_{\text{eval}}$ for deciding near optimality, where $\epsilon_{\text{eval}}$ is the error bound of the applied Chernoff bound (2).

We balance the number of test steps for the incremental and the monolithic approach by executing the same number of tests. As a result, the simulation costs for executing tests is approximately the same. Since the incremental approach requires model learning and model checking in each round, it will also require more computation time than the monolithic approach. While our main focus is on evaluating with respect to the achieved probability estimation, we will briefly discuss computation cost at the end of the section.

We also briefly discuss estimations based on model checking of inferred models $\mathcal{M}_{h,s}$, i.e. $\max_{\phi} P_{\mathcal{M}_{h,s}}(\phi)$ calculated by PRISM [29]. These estimations have also been discussed by Mao et al. [36]. They noted that estimations may differ significantly from optimal values in some cases, but generally represent good approximations.

Implementation and settings  We base the evaluation on our Java implementation of the described technique which can be found at [43]. All experiments were performed with a Lenovo Thinkpad T450 with 16 GB RAM and an Intel Core i7-5600U CPU operating at 2.6 GHz and running Xubuntu Linux 18.04. The systems were modelled with PRISM [29]. PRISM served three purposes:

- We exported the state, transition, and label information from models. We simulated the models in a black-box fashion with this information.
- The maximal probabilities were computed via PRISM.
- PRISM’s scheduler generation was used for scheduler inference.

Simulation, as well as sampling, is controlled by probabilistic choices. To ensure reproducibility, we used fixed seeds for pseudo-random number generators controlling the choices. All experiments were run with 20 different seeds and we discuss statistics derived from 20 such runs. For the evaluation of schedulers, we applied a Chernoff bound with $\epsilon_{\text{eval}} = 0.01$ and $\delta_{\text{eval}} = 0.01$. In contrast to the conference version of this paper [3], we used a fixed significance level for the compatibility check of IOAlergia, by setting $\epsilon_{\text{Alergia}} = 0.5$, a value also used by Mao et al. [36]. They noted that IOAlergia is generally robust with respect to the choice of this value, but we found that our approach benefits from a larger $\epsilon_{\text{Alergia}}$, which causes fewer state merges and consequently larger models. Put differently, our approach benefits from more conservative state merging. As noted in Sect. 4, we aim at ensuring broad exploration in the beginning and property-directed exploration in later
Table 3 Parameter settings for the slot-machine case study

| Parameter    | Value  |
|--------------|--------|
| \( p_{\text{quit}} \) | 0.05   |
| maxRounds   | 100    |
| \( n_{\text{batch}} \) | 1000   |

(a) Reaching \( \text{Pr10} \) in less than 8 steps

(b) Reaching \( \text{Pr10} \) in less than 14 steps

![Diagram](image)

Fig. 3 Simulation-based probability estimations of reaching \( \text{Pr10} \)

rounds. Therefore, we set \( p_{\text{start}} = 0.75 \) and \( c_{\text{change}} = 0.95 \) unless otherwise noted. We set the convergence-check parameters in all experiments as suggested in Sect. 4: \( \alpha_{\text{conv}} = 0.01 \), \( \epsilon_{\text{conv}} = 0.01 \), and \( r_{\text{conv}} = 6 \). Table 2 summarises parameter settings that apply in general.

5.1 Slot machine

The slot machine was analysed in the context of MDP inference before [36]. Basically, it has three reels which are initially blank and which either show apple or bar after spinning (one input per reel). With increasing number of spins the probability of bar decreases. A player is given a number of spins \( m \), after which one of three prizes is awarded depending on the reel configuration. A fourth input leads with equal probability either to two extra spins (with a maximum of \( m \)), or to stopping the game prematurely including issuance of prizes. For the evaluation, we reimplemented the model, therefore probabilities and state space differ from [36]. As property, we investigated reaching the output \( \text{Pr10} \) if \( m = 5 \), representing a prize that is awarded after stopping the game, if all reels show bar. The parameter settings for the learning experiments are given by Table 3, that is, \( p_{\text{quit}} = 0.05 \), maxRounds = 100, and \( n_{\text{batch}} = 1000 \).

Figure 3 shows evaluation results comparing the different approaches. Box plots summarising the probability estimations for reaching \( \text{Pr10} \) in less than 8 steps are shown in Fig. 3a and b shows results for a limit of 14 steps. From left to right, the blue boxes correspond to \( s_{\text{mono}} \), the black boxes correspond to \( s_{\text{inc}} \), and the red boxes correspond to \( s_{\text{conv}} \), i.e. the incremental approach with convergence check. Dashed lines mark optimal probabilities. Note that estimations may be slightly larger than the optimal value in rare cases because they are based on simulations. This can be observed for \( s_{\text{inc}} \) in Fig. 3a and also in some of the following experiments. The applied Chernoff bound gives a confidence value for staying within error bound \( \epsilon_{\text{eval}} \), in case we actually found an optimal scheduler.

Estimations with the baseline \( s_{\text{unif}} \) are fairly constant, at approximately 0.012 for 8 steps and at 0.019 for 14 steps. As estimations with \( s_{\text{mono}} \), \( s_{\text{inc}} \), and \( s_{\text{conv}} \) are significantly higher,
this shows that our approach positively influences the probability of reaching a desired event. We further see that the incremental approach performs better than the monolithic, whereby the gap increases with step size. Unlike the monolithic approach, the incremental approach finds near-optimal schedulers in both cases. However, the relative number of near-optimal schedulers decreases with increasing step bound.

Early stopping via detecting convergence affects performance only slightly. The differences between the quartiles derived for $s_{\text{conv}}$ and for $s_{\text{inc}}$ are actually lower than the error bound $\epsilon_{\text{eval}} = 0.01$. Random variations could therefore be the cause of a visually perceived performance change. For the first experiment with a limit of 8 steps, early stopping reduced the number of executed rounds to 72.2 on average and to 71.05 for the second experiment. However, one run of the first experiment failed to stop early, because convergence was not detected in less than 100 rounds. Runs of the second experiment executed at most 92 rounds.

Alternatively to simulation-based estimation, estimations may be based on model checking an inferred model [36]. For that, a model $M$ is inferred, either incrementally or in a single step, and then a probabilistic model-checker computes $\max_s P_{M,s}(\phi)$. In other words, SMC of the actual SUT controlled by an inferred scheduler is replaced by probabilistic model-checking of a learned model. In the first scenario, estimations are generally bounded above by the optimal probability while estimations in the second scenario may also overestimate the true optimal probability. An advantage of the second scenario is that it reduces the simulation cost since SMC requires additional sampling of the SUT.

Figure 4a and c show model-checking-based estimations of reaching $Pr10$ in less than 8 and 14 steps respectively. Here, $s_{\text{mono}}$ denotes that the models $M$ were inferred in one step, while $s_{\text{inc}}$ denotes incremental model-inference. Incremental model-inference with early stopping is labelled $s_{\text{conv}}$. The figures demonstrate that these estimations differ from estimations obtained via SMC (see Fig. 3). The monolithic approach significantly overestimates in both cases. The incremental approach leads to more accurate results. None of the measurement results exceeds the optimal value by more than $\epsilon_{\text{eval}}$. Note that early stopping did not significantly affect these estimations. Still, the SMC-based estimations are more reliable in the sense that they establish an approximate lower bound for the true optimal probability.

5.2 MQTT with stochastic failures

This case study is based on a Mealy-machine model of an MQTT [8] broker, learned in previous work [49]. We transformed a model of the EMQ [18] broker interacting with two clients, adding stochastic failures to connection acknowledgements and subscription acknowledgements for the second client, whereby we set $p_{\text{cr}} = 0.1$. For the evaluation we
Table 4  Parameter settings for the MQTT case study

| Parameter     | Value                  |
|---------------|------------------------|
| \(p_{\text{quit}}\) | 0.025                  |
| \(\text{maxRounds}\) | 60 (240 for \(s_{\text{conv}}\)) |
| \(n_{\text{batch}}\)       | 100                    |

Fig. 5 MQTT: box plots of probability estimations of different inference approaches

Infer schedulers \(s\) maximising \(\mathbb{P}_{M,s}(F^{<k}\text{crash})\) for \(k \in \{5, 8, 11, 14, 17\}\). For the sampling, we set \(p_{\text{quit}} = 0.025\). While this leads to samples longer than necessary for evaluation, e.g., for \(k = 5\) the expected length of traces is 43, this increases the chance of seeing \(\text{crash}\) in a sample which is reflected in inferred models. The simulation budget is limited by \(\text{maxRounds} = 60\), and \(n_{\text{batch}} = 100\) for the incremental approach without early stopping. Since the experiments required more than 60 rounds for convergence to be detected, we set \(\text{maxRounds}\) to 240 for the incremental approach with convergence check. The parameter are also summarised in Table 4.

Figure 5 shows box plots for the learning-based approaches. At each \(k\), the box plots from left to right summarise measurements for \(s_{\text{mono}}\) (blue), \(s_{\text{inc}}\) (black), and \(s_{\text{conv}}\) (red). The dashed line is the optimal probability achieved with \(s_{\text{opt}}\), and the solid line represents the average probability of reaching \(\text{crash}\) with a uniformly randomised scheduler. The box plots demonstrate that larger probabilities are achievable with learning-based approaches than with random testing. All runs including outliers reach \(\text{crash}\) with a higher probability than random testing. The monolithic approach, however, only performs marginally better in some cases. Both incremental approaches achieve near-optimal results more reliably. All learning-based approaches find at least one near-optimal scheduler out of 20, but incremental inference finds near-optimal schedulers more reliably.

The convergence check causes a reliability gain for \(k = 8\) and \(k = 17\) in this case study, as it basically detected that executing 60 rounds is not enough. It generally required more than 60 rounds to detect convergence, except in a few cases. Experiments for larger values of \(k\) required slightly more rounds to be executed, such that on average 79.6 rounds were executed for \(k = 17\). In contrast to this, we executed on average only 72.15 for \(k = 5\). We also see that most estimations of \(s_{\text{inc}}\) and \(s_{\text{conv}}\) are in a small range near to the optimal values. However, a few outliers are significantly lower, e.g., at 0.46 for \(k = 8\). Therefore, it makes sense to infer multiple schedulers and discard those performing poorly.

Model-checking-based estimations of reaching \(\text{crash}\) with the incremental approach led to overestimations in some cases. For instance, the maximal estimation for \(k = 11\) is 0.724 while 0.651 is the true optimal value, and also for \(k = 5\) one run leads to a model-checking-based estimation of 0.373 although 0.344 is the true optimal value. This is in contrast to the
Table 5 Parameter settings for the TCP case study

| Parameter   | Value             |
|-------------|-------------------|
| $p_{\text{quit}}$ | 0.025             |
| $\maxRounds$ | 120 (240 for $s_{\text{conv}}$) |
| $n_{\text{batch}}$ | 250               |

Fig. 6 TCP: box plots of probability estimations of different inference approaches

slot machine example (see Fig. 4), where the incremental approach produced results close to or lower than the optimal value.

5.3 TCP with stochastic failures

This case study builds upon a Mealy-machine model of Ubuntu’s TCP server learned by Fiterău-Broștean et al. [20,50]. In previous work [2], we have shown that conformance testing of this system is challenging. Here, we consider a version with random crashes with $p_{\text{cr}} = 0.05$, as discussed in the beginning of this section. We mutated transitions to states outputting an acknowledgement which increments both sequence and acknowledgement numbers. For the evaluation, we infer schedulers for $P_{s_{\text{mon}},s}(F < k \text{ crash})$ with $k \in \{5, 8, 11, 14, 17\}$ and we set $\maxRounds = 120$, and $n_{\text{batch}} = 250$ for the incremental approach without early stopping. Consequently, we set $n_{\text{batch}} = 250 \cdot 120$ for the monolithic approach. Since the convergence check detected convergence only after 120 rounds in several experiments, we set $\maxRounds$ to 240 for the incremental approach with early stopping. We set $p_{\text{quit}}$ to the same value as for MQTT. The parameter settings are also shown in Table 5.

Figure 6 shows box plots summarising the collected probability estimations. As before, there are groups of three box plots at each $k$, which from left to right represent $s_{\text{mon}}, s_{\text{inc}},$ and $s_{\text{conv}}$. The figure does not include plots for random testing with $s_{\text{unif}}$, because it reaches the crash with very low probability. Estimations produced by $s_{\text{unif}}$ are lower than 0.01 for all $k$. This demonstrates that random testing is insufficient in this case to reliably reach crashes of the system.

We further see that all learning-based approaches achieve to generate near-optimal schedulers for all $k$. As before, both configurations of the incremental approach are more reliable than the monolithic approach. For this more complex system, the reliability gain from incremental scheduler generation is actually much larger than for the MQTT experiments. Early stopping affects probability estimations only marginally. This is also in line with previous observations.
Like for MQTT, we needed to set \textit{maxRounds} to a value larger than initially planned, for convergence to be detected. There is a large spread in the number of executed rounds, e.g., we executed between 42 and 240 rounds for \( k = 14 \). In this case, convergence was detected after 133.5 rounds on average. The average number of executed rounds is lower than 135 rounds for all \( k \).

5.4 Gridworld

The following case study is inspired by a motion-planning scenario discussed by Fu and Topcu [23], also in the context of learning control strategies. In the experiments, we generate schedulers for a robot navigating in a gridworld environment. These schedulers shall with high probability reach a fixed goal location after starting from a fixed initial location.

A gridworld consists of tiles of different terrains and is surrounded by walls. To model obstacles, interior tiles may be walls as well. The robot starts at a predefined location and may move into one of four directions, i.e. we select from four inputs. It can observe changes in the type of terrain, whether it bumped into a wall, and whether it is located at the goal location. If the robot bumps into a wall, it will not change location. Whenever the robot moves, it may not reach its target, but rather reach a neighbouring tile with some probability, unless the neighbouring tile is a wall. That is, if the robot moves north, it may reach the tile situated north west or north east to its original position. The probability of such an error depends on the terrain of the target tile. We distinguish the terrains (with error probabilities in parentheses): \textit{Mud} (0.4), \textit{Sand} (0.25), \textit{Concrete} (0), and \textit{Grass} (0.2). As indicated above, \textit{Wall} is actually also a terrain that cannot be entered.

Figure 7a shows the gridworld, we used for evaluation. Black tiles represent walls, while the other terrains are represented by different shades of grey and their initial letters. A circle marks the initial location and a double circle marks the goal location. Although its state space, containing \( 35^2 \) different states, is relatively small, navigating in this gridworld is challenging without prior knowledge. Initially, three moves to the right are necessary, as walls block direct moves towards the goal. This mimics the requirement of an initialisation routine.

Before discussing measurements, we want to briefly describe the structure of the MDP modelling this gridworld and how probabilities affect it. The initial state is labelled \( C \) and corresponds to the location with the coordinate \((1, 1)\). Moving towards north is not possible, therefore the input \textit{north} leads to a state labelled \textit{wall}, which also corresponds to the coordinate \((1, 1)\). If the robot instead moves towards east, it will reach a state corresponding to the coordinate \((2, 1)\), which is labelled \( C \). Moving from coordinate \((3, 1)\) towards east, the target tile with the coordinate \((4, 1)\) is labelled by \( M \) (Mud), which has a non-zero error probability of 0.4. Therefore, the input \textit{east} causes a stochastic transition; with a probability of 0.6 the robot reaches the target coordinate \((4, 1)\) and observes \( M \), but with a probability of 0.4, it reaches the location \((4, 2)\) to the south and observes \( C \).

To infer schedulers, we applied the configuration given by Table 6, that is, \textit{maxRounds} = 150, and \( n_{\text{batch}} = 500 \), and \( p_{\text{quit}} = 0.5 \). Due to the larger value of \textit{maxRounds}, we increased \( c_{\text{change}} \) as well to 0.975. This causes more random choices and thereby broad exploration in a larger number of rounds. As this case study differs significantly from the others, we chose \textit{maxRounds} conservatively, performing a larger number of rounds.

---

2 The number of states does not equal the number of reachable locations because locations adjacent to walls require two states in the MDP – one outputting the terrain and one with the output \textit{wall}. States outputting \textit{wall} are reached after the robot bumps into a wall.
Figure 7 shows measured estimations of $P_{M,s}(F < 10)_{\text{goal}}$ for $s_{\text{inc}}$, $s_{\text{conv}}$, $s_{\text{mono}}$, and random testing with $s_{\text{unif}}$. The dashed line denotes the optimal probability.

Random testing obviously fails to reach the goal in less than ten steps. This is caused by the fact that it is unlikely to navigate past the walls via random exploration. The performance of the monolithic approach is also affected by this issue, because it learns solely from uniformly randomised sample traces. Random exploration covers only the initial part of the state space thoroughly. Therefore, the monolithically generated schedulers tend to perform worse than incrementally generated. By directing exploration towards the goal, the incremental approach achieves to generate near-optimal schedulers.

We also see that the impact of the convergence check is not severe. Both settings, with and without convergence check, produced similar results. The convergence check was able to reduce simulation costs for all but three runs of the experiment, in which convergence was not detected in less than 150 rounds. The incremental scheduler generation required at least 94 rounds and on average 131.9 rounds were executed before convergence was detected.

### 5.5 Shared coin consensus

The last case study examines scheduler generation for a randomised consensus protocol by Aspnes and Herlihy [6]. In particular, we used a model of the protocol distributed with the PRISM model checker [29] as a basis for this case study.\(^3\) Note that we did not change the functionality of the protocol, but only performed minor adaptations such as adding action labels for inputs.

This protocol’s goal is to achieve consensus between multiple asynchronous processes, i.e. to find a preferred value of either 1 or 2 on which all processes agree. In the model distributed

---

\(^3\) A thorough discussion of the model and related experiments can be found at [http://www.prismmodelchecker.org/casestudies/consensus_prism.php](http://www.prismmodelchecker.org/casestudies/consensus_prism.php). Accessed: 2018-12-03
Table 7  Parameter settings for the consensus-protocol case study

| Parameter     | Value |
|---------------|-------|
| $p_{\text{quit}}$ | 0.025 |
| maxRounds     | 100   |
| $n_{\text{batch}}$ | 250   |

with PRISM, the processes share a global counter $c$ with a range of $[0..2\cdot(K+1)\cdot N]$ where $N$ is the number of processes and $K$ is an integer constant. Initially, $c$ is set to $K + 1\cdot N$. All involved processes perform the following steps to locally determine a preferred value $v$:

1. Flip a fair coin (local to the process)
2. Check coin
   2.1 If the coin shows tails, decrement shared counter $c$
   2.2 Otherwise increment $c$
3. Check value of $c$
   3.1 If $c \leq N$, then the preferred value is $v = 1$
   3.2 If $c \geq 2 \cdot (K + 1) \cdot N - N$, then $v = 2$
   3.3 Otherwise goto 1.

Each of those actions, flipping a coin, checking it and checking the value of $c$ represents one step in the protocol. Since the processes execute asynchronously, their actions may be arbitrarily interleaved, whereby the interleavings are controlled by schedulers. A schedulers may choose from $N$ inputs $go_i$, one for each process $i$. Performing $go_i$ basically instructs process $i$ to perform the next step in the protocol. If process $i$ already picked a preferred value in Step 3.1. or in Step 3.2., $go_i$ is simply ignored.

The visible outputs of the system are sets of propositions that hold in the current step. Firstly, the propositions expose the current value of the shared counter, i.e. they include $(c = k)$ for $a k \in [0..2\cdot(K+1)\cdot N]$. Secondly, they expose values of the local coins, i.e. the outputs include one $(\text{coin}_i = x)$ for each process $i$, where $x \in \{\text{heads}, \text{tails}\}$. Additionally, the outputs may include a proposition finished, signalling that processes decided on a preferred value. As generating schedulers for this protocol in a learning-based fashion represents a demanding task, we only consider the case of $K = 2$ and $N = 2$, i.e. two asynchronously executing processes. Setting either of these constants to larger values significantly increases the number of steps to reach consensus.

Note that information about the current value of local coins is necessary to be able to generate optimal schedulers. Consider the property $\phi = F_c < 5 c = 5$ and $\max_s P_{M,s}(\phi)$, which is equal to 0.75, for that. Initially, we have $c = 6$ and an optimal scheduler may choose any action, say go$_1$. After that, we have coin$_1 = \text{heads}$ with 0.5 probability and we should perform go$_2$, because go$_1$ would increment $c$. After performing go$_2$, we have coin$_2 = \text{heads}$ with 0.5 probability and cannot satisfy $\phi$ anymore. All other traces would satisfy $\phi$. Without knowledge about the state of local coins, we would not be able to make sensible choices of inputs. The randomised state machines controlling the processes remain a black box to us, though. Models of their composition are inferred via learning.

For the measurements, we optimise $P_{M,s}(F_{<k}\text{finished})$ for $k \in \{14, 20\}$, i.e. we try to find schedulings of the two processes which optimise the probability of finishing the protocol in less than 14 and 20 steps, respectively. Finishing here means that both processes picked a preferred value. In the experiments, we applied the configuration given by Table 7, that

\[\textit{Table 7} \quad \text{Parameter settings for the consensus-protocol case study}\]

| Parameter     | Value |
|---------------|-------|
| $p_{\text{quit}}$ | 0.025 |
| maxRounds     | 100   |
| $n_{\text{batch}}$ | 250   |
is, $maxRounds = 100$, and $n_{batch} = 250$, and $p_{quit} = 0.025$. Since we know that states outputting $finished$ are absorbing, we stopped sampling upon seeing the $finished$ output, as suggested in [35].

Figure 8 shows evaluation results for the incremental and the monolithic approach in comparison to random testing. The box plots corresponding to each of these are labelled $s_{mono}$, $s_{inc}$ and $s_{unif}$, respectively. The dashed line represents the optimal probability as before. In contrast to previous experiments, we see that the monolithic approach may perform worse than random testing. For $k = 14$, there are three measurements, which are exactly zero, but more than a quarter of the measurement results are near-optimal. For $k = 20$, the number of experiments achieving lower estimations than random testing decreases to two, but none of the generated schedulers is near-optimal. This can be explained by considering the minimum number of steps necessary to reach $finished$. We need to execute at least 12 steps to observe $finished$. As a result, it may happen that relevant parts of the system, states reached only after 12 steps, are inaccurately learned. This exemplifies that incremental scheduler generation pays off, because it is able to generate near-optimal schedulers for both values of $k$. For $k = 14$, three quarter of the incrementally generated schedulers are near-optimal and for $k = 20$, more than one quarter of the schedulers are near-optimal.

This case study actually highlights a weakness of our convergence check. It assumes that the search will converge to some unique behaviour. The protocol is completely symmetric for both processes, so it does not matter which process performs the first step. Hence, there are at least two optimal schedulers which differ in their initial action. This action is present in each of the 250 traces collected in one round, which presumably include further ambiguous choices. This causes SIMILARSCHED to return $false$ in most of the cases. Consequently, we do not discuss results obtained with the convergence check, as it rarely led to early stopping. A possible approach to counter this problem would be, assuming there is a lexicographic ordering on inputs, to always select the lexicographically minimal input, in case the choice is ambiguous.

### 5.6 Convergence check

We discussed the influence and application of early stopping throughout this section. Now, we want to briefly examine the underlying assumption. The rationale behind the convergence check and early stopping is that scheduler behaviour converges with increasing number of rounds. As a result, fluctuations in the probability estimations produced by schedulers are expected to diminish. Ideally, estimations should increase over time as well, i.e. schedulers
should improve. To investigate whether our assumptions hold, we applied the incremental approach, evaluated intermediate schedulers, and collected the produced probability estimations.

Figure 9 contains graphs showing statistics summarising the collected estimations. The experiment summarised in Fig. 9a optimises reaching $Pr10$ in less than 14 steps with the slot machine. Figure 9b shows statistics for reaching goal in the gridworld in less than 10 steps. The graphs read as follows: the horizontal axis displays the rounds, and the vertical axis displays the value of the probability estimations. The lines from top to bottom represent the maximum, the third quartile, the median, the first quartile, and the minimum computed from the estimations collected in each round. Like before, these values were computed from 20 runs.

In both cases, we see that fluctuations decrease over time. The interquartile range decreases as well until it becomes relatively stable. Stable estimations are reached at around the 70th round in Fig. 9a, which is the area where convergence was detected – we stopped on average after 71.05 rounds. We see larger fluctuations of the minimal value in Fig. 9b, but they decline as well. Fluctuations of the minimal value can also be observed after 150 rounds. As a result, we may stop too early in rare cases.

Figure 9b also reveals unexpected behaviour. Testing of the gridworld actually required relatively few rounds of learning to achieve good results. In particular, the estimations after the first rounds were larger than expected, because the basis for the first round of learning is formed by only $n_{\text{batch}}$ random tests.
Table 8  Average runtime of learning and scheduler generation for various properties (all values in seconds)

| Case study and property | Operation     | $s_{\text{mono}}$ | $s_{\text{inc}}$ | $s_{\text{conv}}$ |
|-------------------------|---------------|-------------------|-----------------|------------------|
| Slot machine: $F^{<14}_{Pr10}$ | Learning      | 3.4              | 114.4           | 64.8             |
|                         | Scheduler generation | 3.8            | 364.6           | 226.6            |
| MQTT: $F^{<17}_{\text{crash}}$ | Learning      | 1.7              | 50.5            | 75.4             |
|                         | Scheduler generation | 2.6            | 166.4           | 174.0            |
| TCP: $F^{<17}_{\text{crash}}$ | Learning      | 21.3             | 471.7           | 564.4            |
|                         | Scheduler generation | 3.4            | 256.9           | 248.3            |
| Gridworld: $F^{<10}_{\text{goal}}$ | Learning      | 2.1              | 97.5            | 96.7             |
|                         | Scheduler generation | 2.0            | 324.6           | 317.9            |
| Shared coin: $F^{<20}_{\text{finished}}$ | Learning      | 7.7              | 300.9           | –                |
|                         | Scheduler generation | 3.9            | 316.8           | –                |

5.7 Runtime

We simulated previously learned models for the evaluation, thus the simulation cost was very low in our experiments. As a result, the time for learning and reachability checking dominated the computation time. Table 8 lists the average runtime of learning and reachability checking for the property with the largest bound of each case study.

It can be seen that the incremental approaches, denoted by $s_{\text{inc}}$ and $s_{\text{conv}}$, require considerably more time to complete. Incremental scheduler generation without convergence detection for instance takes on average 728.6 s for the TCP property $F^{<17}_{\text{crash}}$, while the monolithic approach requires only 24.7 seconds. Thus, the better performance with respect to maximizing probability estimations comes at the cost of increased runtime for learning and scheduler generation. In a testing scenario with real-world implementations, however, this time overhead may be negligible. If network communication is necessary, e.g., for protocol testing, the simulation time required for interacting with the SUT can be assumed to dominate the overall runtime. To contrast simulation runtime to the runtime of learning and scheduler generation, consider the hypothetical, but realistic scenario in which each simulation step takes about 10 milliseconds. Both approaches, the monolithic and the incremental, require approximately the same number of simulation steps, about $1.7 \cdot 10^6$ for $F^{<17}_{\text{crash}}$. In this scenario, the simulation duration would amount to about 4.7 h, such that the runtime overhead of 703.9 s caused by the incremental approach would be low in comparison. Similar observations can be made for other case studies. Since the time spent simulating the SUT can be expected to dominate the computation time, we conclude that the incremental approach is preferable to the monolithic approach in this context.

In Table 8, we also see that the convergence detection provides a performance gain for the slot machine, but causes slightly worse runtime for MQTT and TCP. The decreased performance is caused by the fact that convergence often could not be detected within the $\text{maxRounds}$ used for $s_{\text{inc}}$. Therefore, we increased $\text{maxRounds}$ of $s_{\text{conv}}$ for MQTT and TCP, as discussed above. However, the goal of convergence detection is not a reduction of runtime. With the convergence detection heuristic, we want to provide a stopping criterion that does not solely rely on an arbitrarily chosen $\text{maxRounds}$ parameter.

Finally, we want to discuss the runtime complexity of learning and scheduler generation. The worst-case complexity of IOALERGIA is cubic in the size of the merged tree-shaped
representation of the sampled traces, but the typical behaviour observed in practice is approximately linear [36]. Hence, it is unlikely that learning runtime could be improved, but the scheduler generation runtime can potentially be improved. Our implementation communicates with PRISM via files, standard input and standard output. As a result, there is substantial communication overhead that could be removed via a tighter integration of scheduler generation. PRISM’s default technique for scheduler generation, which is called value iteration, could also benefit algorithmically from such a tight integration. Since we check reachability with respect to a bound $k$, we could also bound the number of iterations performed by value iteration by $k$ [7]. This leads to a worst-case runtime complexity of $O(k \cdot n^2 \cdot m)$, where $n$ is the number of states of a learned model and $m$ is the number of inputs. The number of inputs $m$ is generally a small constant and we have observed that $n^2$ is generally smaller than the number of sampling steps required for learning. As a result, we expect the simulation time to generally dominate in non-simulated scenarios.

5.8 Discussion

We applied our approach for various types of models in several configurations, which we compared among each other, to the true optimal testing strategy and also to random testing as a baseline. The results of the performed experiments show (1) that learning-based approaches outperform the baseline and (2) that the incremental approach is able to generate near-optimal schedulers in all cases. In most experiments, the median probability estimation derived with the incremental approach was near-optimal, thus it generated near-optimal schedulers reliably. We have also seen that the convergence detection heuristic did not have a negative impact on the accuracy of the incremental approach. However, we are not able to give concrete bounds on the required number of samples to achieve a desired success rate. This is due to the fact that we rely on IOALERGIA, for which convergence in the limit has been shown [36], but stronger guarantees are currently not available.

We generally targeted systems with small state space. An application in practice therefore requires abstraction to ensure that the state space is not prohibitively large. This is generally required in learning-based verification and several applications have shown that learning combined with abstraction provides effective means to enable model-based analysis [20,21,44,49].

In addition to that, we have also seen limitations that cannot be solved by abstraction. The differences between estimations and optimal values tend to increase with the step bound $k$. This is potentially caused by the exponential growth of different traces. This growth also affects the application of the approach for large gridworld examples. Increasing the width of the gridworld also increases the steps required to reach the goal and causes the performance to drop. A possible mitigation would be to identify disabled inputs, i.e. inputs rejected by the SUT, if we have such information. This might prevent certain traces from being executed beyond a disabled input. In the original version of IOALERGIA [35], such knowledge facilitates learning, because disabled inputs are assumed to leave the current state unchanged. In the gridworld example, we may consider inputs to be disabled, if they cause the robot to move into a wall.

A related issue also affects the case study on the consensus protocol. Changing the number of processes from two to four, increases the minimum number of steps to reach finished to 24. Here, composition actually causes the state space to grow. We could tackle such problems via decomposition. Instead of learning a large monolithic model, several small models could be learned, which would then be composed for the reachability analysis.
6 Conclusion

We presented an approach to infer near-optimal schedulers for reachability objectives of MDPs. To our knowledge, it is the first such approach to be applicable in a purely black-box setting, where only the input interface is known. This is accomplished by incrementally refining the knowledge about the system, via model inference and based on that property-directed exploration of the system. Section 5 presents promising results, showing that near-optimality can be achieved.

Therefore, we plan to investigate this approach further and extend it. As a first step, we are currently evaluating the method on more case studies. In order to be able to examine more complex systems, we are planning to work on compositional verification. That is, we are investigating how to benefit from decomposition, as opposed to treating composed systems as large monolithic systems. We are also studying the applicability in different testing scenarios. In a testing context, e.g., Nachmanson et al. [38] discussed strategies for bounded reachability games, but with a given model.

Furthermore, non-functional properties like execution time could be considered. For that, we need to devise a model-inference technique which considers both non-deterministic choice of inputs and (continuous) time. Current approaches for probabilistic timed systems do not account for non-determinism of this form [36,52]. If we had such a technique, we could, e.g., use PRISM with the digital clocks engine [29] or UPPAAL Stratego [15,16] to infer schedulers. Another possible extension would be to consider more general properties than reachability. This would require replacing PRISM’s scheduler generation in our approach. In conclusion, we believe that our results are encouraging and that there are many promising directions for future research.

Acknowledgements Open access funding provided by Graz University of Technology. This work was supported by the TU Graz LEAD project “Dependable Internet of Things in Adverse Environments”. The authors would like to thank the LEAD project members Roderick Bloem, Masoud Ebrahimi, Franz Pernkopf, Franz Röck, and Tobias Schrank for fruitful discussions.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Aichernig BK, Mostowski W, Mousavi MR, Tappler M, Taromirad M (2018) Model learning and model-based testing. In: Bennaceur A, Hähnle R, Meinke K (eds) Machine learning for dynamic software analysis: potentials and limits—international Dagstuhl seminar 16172, Dagstuhl Castle, Germany, April 24–27, 2016. Revised papers, Lecture notes in computer science, vol 11026, pp 74–100. Springer. https://doi.org/10.1007/978-3-319-96562-8_3
2. Aichernig BK, Tappler M (2017) Learning from faults: mutation testing in active automata learning. In: Barrett C, Davies M, Kahsai T (eds) NASA formal methods—9th international symposium, NFM 2017, Moffett Field, CA, USA, May 16–18, 2017. Proceedings, Lecture notes in computer science, vol 10227, pp 19–34. https://doi.org/10.1007/978-3-319-57288-8_2
3. Aichernig BK, Tappler M (2017) Probabilistic black-box reachability checking. In: Lahiri SK, Reger G (eds) Runtime verification—17th international conference, RV 2017, Seattle, WA, USA, September 13–16, 2017. Proceedings, Lecture notes in computer science, vol 10548, pp 50–67. Springer. https://doi.org/10.1007/978-3-319-67531-2_4
4. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf. Comput. 75(2):87–106. https://doi.org/10.1016/0890-5401(87)90052-6
5. Argyros G, Stais I, Jana S, Keromytis AD, Klavias A (2016) SFADiff: automated evasion attacks and fingerprinting using black-box differential automata learning. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp 1690–1701. ACM. https://doi.org/10.1145/2976749.2978383

6. Aspnes J, Herlihy M (1990) Fast randomized consensus using shared memory. J Algorithms 11(3):441–461. https://doi.org/10.1016/0196-6774(90)90021-6

7. Baier C, Katoen J (2008) Principles of model checking. MIT Press, Cambridge

8. Banks A, Gupta, R (ed.) (2014) MQTT version 3.1.1. OASIS standard. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

9. Beimel A, Bergadano F, Kushilevitz E, Varricchio S (2000) Learning functions represented as multiplicity automata. J ACM 47(3):506–530. https://doi.org/10.1145/337244.337257

10. Brázdil T, Chatterjee K, Chmelik M, Forejt V, Kretinsky J, Kwiatkowska MZ, Parker D, Ujma M Verifi-
cation of Markov decision processes using learning algorithms. In: Cassez and Raskin [12], pp 98–114. https://doi.org/10.1007/978-3-319-11936-6_8

11. Carrasso RC, Oncina J (1994) Learning stochastic regular grammars by means of a state merging method. In: Carrasso RC, Oncina J (eds) Grammatical inference and applications, second international colloquium, ICGL-94, Alicante, Spain, September 21–23, 1994. Proceedings, Lecture notes in computer science, vol 862, pp 139–152. Springer. https://doi.org/10.1007/3-540-58473-0_144

12. Cassez F, Raskin J (eds) (2014) Automated technology for verification and analysis–12th international symposium, ATVA 2014, Sydney, NSW, Australia, November 3–7, 2014. Proceedings, Lecture notes in computer science, vol 8837. Springer. https://doi.org/10.1007/978-3-319-31936-6

13. Chen Y, Nielsen TD (2012) Active learning of Markov decision processes for system verification. In: 11th international conference on machine learning and applications, ICMLA, Boca Raton, FL, USA, December 12–15, 2012, vol 2, pp 289–294. IEEE. https://doi.org/10.1109/ICMLA.2012.158

14. D’Argenio P, Legay A, Sedwards S, Traonouez L (2015) Smart sampling for lightweight verification of Markov decision processes. STTT 17(4):469–484. https://doi.org/10.1007/s10009-015-0383-0

15. David A, Jensen PG, Larsen KG, Legay A, Lime D, Sørensen MG, Taankvist JH. On time with minimal expected cost! In: David A, Jensen PG, Larsen KG, Legay A, Lime D, Sørensen MG, Taankvist JH. On time with minimal expected cost! In: Cassez and Raskin [12], pp 129–145. https://doi.org/10.1007/978-3-319-11936-6_10

16. Elkind E, Genest B, Peled DA, Qu H (2006) Grey-box checking. In: Elkind E, Genest B, Peled DA, Qu H (eds) Grey-box checking. In: Najm E, Pradat-Peyre J, Donzeau-Gouge V (eds) Formal techniques for networked and distributed systems–FORTE 2006, 26th IFIP WG 6.1 international conference, Paris, France, September 26–29, 2006. Lecture notes in computer science, vol 4229, pp 420–435. Springer. https://doi.org/10.1007/11888116_30

17. Fiterau-Brostean P, Janssen R, Vaandrager FW (2016) Combining model learning and model checking to analyze TCP implementations. In: Chaudhuri S, Farzan A (eds) Computer aided verification–28th international conference, CAV 2016, Toronto, ON, Canada, July 17–23, 2016. Proceedings, Part II, Lecture notes in computer science, vol 9780, pp 511–521. Springer. https://doi.org/10.1007/978-3-642-43721-2_40

18. Fiterau-Brostean P, Janssen R, Vaandrager FW (2016) Combining model learning and model checking to analyze TCP implementations. In: Chaudhuri S, Farzan A (eds) Computer aided verification–28th international conference, CAV 2016, Toronto, ON, Canada, July 17–23, 2016. Proceedings, Part II, Lecture notes in computer science, vol 9780, pp 511–521. Springer. https://doi.org/10.1007/978-3-642-43721-2_40

19. Fiterau-Brostean P, Lenaerts T, Poll E, de Ruiter J, Vaandrager FW, Verleg P (2017) Model learning and model checking of SSH implementations. In: Erdogmus H, Havelund K (eds) Proceedings of the 24th ACM SIGSOFT international symposium on model checking of software, Santa Barbara, CA, July 10–14, 2017, pp 142–151. ACM. https://doi.org/10.1145/3092282.3092289. http://doi.acm.org/10.1145/3092282.3092289

20. Forejt V, Kwiatkowska MZ, Norman G, Parker D (2011) Automated verification techniques for probabilis-
tic systems. In: Bernardo M, Issarny V (eds) Formal methods for eternal networked software systems–11th international school on formal methods for the design of computer, communication and software systems, SFM 2011, Bertinoro, Italy, June 13–18, 2011. Advanced lectures, Lecture notes in computer science, vol 6659, pp 53–113. Springer. https://doi.org/10.1007/978-3-642-21455-4_3
23. Fu J, Topcu U (2014) Probably approximately correct MDP learning and control with temporal logic constraints. In: Fox D, Kavraki LE, Kurniawati H (eds) Robotics: science and systems X, University of California, Berkeley, July 12–16, 2014. http://www.roboticsproceedings.org/rss10/p39.html
24. Giantamidis G, Tripakis S (2016) Learning Moore machines from input-output traces. In: Fitzgerald JS, Heitmeyer CL, Gnesi S, Philippou A (eds) FM 2016: formal methods 21st international symposium, Limassol, Cyprus, November 9–11, 2016. Proceedings, Lecture notes in computer science, vol 9995, pp 291–309. https://doi.org/10.1007/978-3-319-48989-6_18
25. Grinchtein O, Jonsson B, Leucker M (2004) Learning of event-recording automata. In: Lakhnech Y, Yovine S (eds) Formal techniques, modelling and analysis of timed and fault-tolerant systems, joint international conferences on formal modelling and analysis of timed systems, FORMATS 2004 and formal techniques in real-time and fault-tolerant systems, FTRTFT 2004, Grenoble, France, September 22–24, 2004. Proceedings, Lecture notes in computer science, vol 3253, pp 379–396. Springer. https://doi.org/10.1007/978-3-540-30206-3_26
26. Groce A, Peled DA, Yannakakis M (2002) Adaptive model checking. In: Katoen J, Stevens P (eds) Tools and algorithms for the construction and analysis of systems. In: 8th international conference, TACAS 2002, held as part of the joint European conference on theory and practice of software, ETAPS 2002, Grenoble, France, April 8–12, 2002. Proceedings, Lecture notes in computer science.
27. de la Higuera C (2010) Grammatical inference: learning automata and grammars. Cambridge University Press, New York, NY
28. Khalili A, Tacchella A (2014) Learning nondeterministic Mealy machines. In: Clark A, Kanazawa M, Yoshinaka R (eds) Proceedings of the 12th international conference on grammatical inference, ICGI 2014, Kyoto, Japan, September 17–19, 2014. JMLR workshop and conference proceedings, vol 34, pp 109–123. http://jmlr.org/proceedings/papers/v34/khalili14a.html
29. Kwiatkowska MZ, Norman G, Parker D (2011) PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan G, Qadeer S (eds) Computer aided verification–23rd international conference, CAV 2011, Snowbird, UT, July 14–20, 2011. Proceedings, Lecture notes in computer science, vol 6806, pp 585–591. Springer. https://doi.org/10.1007/978-3-642-22110-1_47
30. Kwiatkowska MZ, Parker D (2013) Automated verification and strategy synthesis for probabilistic systems. In: Hung DV, Ogawa M (eds) Automated technology for verification and analysis–11th international symposium, ATVA 2013, Hanoi, Vietnam, October 15–18, 2013. Proceedings, Lecture notes in computer science, vol 8172, pp 5–22. Springer. https://doi.org/10.1007/978-3-319-02444-8_2
31. Larsen KG, Legay A (2016) Statistical model checking: past, present, and future. In: Margaria T, Steffen B (eds) Leveraging applications of formal methods, verification and validation: foundational techniques–7th international symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10–14, 2016. Proceedings, Part I, Lecture notes in computer science, vol 9952, pp 3–15. https://doi.org/10.1007/978-3-319-47166-2_1
32. Legay A, Delahaye B, Bensalem S (2010) Statistical model checking: an overview. In: Barringer H, Falcone Y, Finkbeiner B, Havelund K, Lee I, Pace GI, Rosu G, Sokolsky O, Tillmann N (eds) Runtime verification–first international conference, RV 2010, St. Julians, Malta, November 1–4, 2010. Proceedings, Lecture notes in computer science, vol 6418, pp 122–135. Springer. https://doi.org/10.1007/978-3-642-16612-9_11
33. Legay A, Seward S, Traonouez L (2014) Scalable verification of Markov decision processes. In: Canal C, Idani A (eds) Software engineering and formal methods–SEFM 2014 collocated workshops: HOFM, SAFORME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, September 1–2, 2014. Revised selected papers, Lecture notes in computer science, vol 8938, pp 350–362. Springer. https://doi.org/10.1007/978-3-319-15201-1_23
34. Mao H, Chen Y, Jaeger M, Nielsen TD, Larsen KG, Nielsen B (2011) Learning probabilistic automata for model checking. In: Eighth international conference on quantitative evaluation of systems, QEST 2011, Aachen, 5–8 September, 2011, pp 111–120. IEEE Computer Society. https://doi.org/10.1109/QEST.2011.21
35. Mao H, Chen Y, Jaeger M, Nielsen TD, Larsen KG, Nielsen B (2012) Learning Markov decision processes for model checking. In: Fahrenberg U, Legay A, Thrane CR (eds) Proceedings quantities in formal methods, QFM 2012, Paris, France, 28 August 2012. EPTCS, vol 103, pp 49–63. https://doi.org/10.4204/EPTCS.103.6
36. Mao H, Chen Y, Jaeger M, Nielsen TD, Larsen KG, Nielsen B (2016) Learning deterministic probabilistic automata from a model checking perspective. Mach Learn 105(2):255–299. https://doi.org/10.1007/s10994-016-5565-9
37. Margaria T, Niese O, Raffelt H, Steffen B (2004) Efficient test-based model generation for legacy reactive systems. In: Ninth IEEE international high-level design validation and test workshop 2004, Sonoma
38. Nachmanson L, Veanes M, Schulte W, Tillmann N, Grieskamp W (2004) Optimal strategies for testing nondeterministic systems. In: Avrunin GS, Rothermel G (eds) Proceedings of the ACM/SIGSOFT international symposium on software testing and analysis, ISSTA 2004, Boston, MA, USA, July 11–14, 2004, pp 55–64. ACM. https://doi.org/10.1145/1007512.1007520

39. Nouri A, Raman B, Bozga M, Legay A, Bensalem S (2014) Faster statistical model checking by means of abstraction and learning. In: Bonakdarpour B, Smolka SA (eds) Runtime verification–5th international conference, RV 2014, Toronto, ON, Canada, September 22–25, 2014. Proceedings, Lecture notes in computer science, vol 8734, pp 340–355. Springer. https://doi.org/10.1007/978-3-319-11164-3_28

40. Okamoto M (1959) Some inequalities relating to the partial sum of binomial probabilities. Ann Inst Stat Math 10(1):29–35. https://doi.org/10.1007/BF02883985

41. Oncina J, Garcia P (1992) Identifying regular languages in polynomial time. In: Advances in structural and syntactic pattern recognition. Volume 5 of series in Machine perception and artificial intelligence, pp 99–108. World Scientific

42. Peled DA, Vardi MY, Yannakakis M (1999) Black box checking. In: Wu J, Chanson ST, Gao Q (eds) Formal methods for protocol engineering and distributed systems, FORTE XII/PSTV XIX’99, IFIP TC6 WG6.1 joint international conference on formal description techniques for distributed systems and communication protocols (FORTE XII) and protocol specification, testing and verification (PSTV XIX), October 5–8, 1999, Beijing, China. IFIP conference proceedings, vol 156, pp 225–240. Kluwer

43. prob-black-reach—Java implementation of probabilistic black-box reachability checking. https://github.com/mtappler/prob-black-reach . Accessed 3 Dec 2018

44. de Ruiter J, Poll E (2015) Protocol state fuzzing of TLS implementations. In: Jung J, Holz T(eds) 24th USENIX security symposium, USENIX Security 15, Washington, D.C., USA, August 12–14, 2015, pp 193–206. USENIX Association. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

45. Sen K, Viswanathan M, Agha G (2004) Statistical model checking of black-box probabilistic systems. In: Alur R, Peled DA (eds) Computer aided verification, 16th international conference, CAV 2004, Boston, MA, USA, July 13–17, 2004. Proceedings, Lecture notes in computer science, vol 3114, pp 202–215. Springer. https://doi.org/10.1007/978-3-540-27813-9_16

46. Shahbaz M, Groz R (2009) Inferring Mealy machines. In: Cavalcanti A, Dams D (eds) FM 2009: formal methods, second world congress, Eindhoven, The Netherlands, November 2–6, 2009. Proceedings, Lecture notes in computer science, vol 5850, pp 207–222. Springer. https://doi.org/10.1007/978-3-642-05089-3_14

47. Shu G, Lee D (2007) Testing security properties of protocol implementations—a machine learning based approach. In: 27th IEEE international conference on distributed computing systems (ICDCS 2007), June 25–29, 2007, Toronto, Ontario, Canada, p 25. IEEE Computer Society. https://doi.org/10.1109/ICDCS.2007.147

48. Sivakorn S, Argyros G, Pei K, Keromytis AD, Jana S (2017) HVLearn: automated black-box analysis of hostname verification in SSL/TLS implementations. In: SP 2017, pp 521–538. IEEE Computer Society. https://doi.org/10.1109/SP.2017.46

49. Tappler M, Aichernig BK, Bloem R (2017) Model-based testing IoT communication via active automata learning. In: 2017 IEEE international conference on software testing, verification and validation, ICST 2017, Tokyo, Japan, March 13–17, 2017, pp 276–287. IEEE Computer Society. https://doi.org/10.1109/ICST.2017.32

50. TCP models. https://gitlab.science.ru.nl/pifteraubrostean/tcp-learner/tree/cav-aec/models . Accessed 3 Dec 2018

51. Utting M, Pretschner A, Legeard B (2012) A taxonomy of model-based testing approaches. Softw Test Verif Reliab 22(5):297–312. https://doi.org/10.1002/stvr.456

52. Verwer S, de Weerd M, Witteveen C (2010) A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data. In: Sempere JM, García P (eds) Grammatical inference: theoretical results and applications, 10th international colloquium, ICGI 2010, Valencia, Spain, September 13–16, 2010. Proceedings, Lecture notes in computer science, vol 6339, pp 203–216. Springer. https://doi.org/10.1007/978-3-642-15488-1_17

53. Volpato M, Tretmans J (2015) Approximate active learning of nondeterministic input output transition systems. In: ECEASST, vol 72. http://journal.ub.tu-berlin.de/eceasst/article/view/1008

54. Wang J, Sun J, Qin S V erifying complex systems probabilistically through learning, abstraction and refinement. In: CoRR. arXiv:1610.06371 (2016)

55. Younes HLS (2005) Probabilistic verification for “black-box” systems. In: Etessami K, Rajamani SK (eds) Computer aided verification, 17th international conference, CAV 2005, Edinburgh, Scotland, July
Publisher's Note  Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.