International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-018-0492-7

FASE 2017

Learning probabilistic models for model checking: an evolutionary approach and an empirical study

Jingyi Wang[1] · Jun Sun[1] · Qixia Yuan[2] · Jun Pang[2]

Abstract

Many automated system analysis techniques (e.g., model checking, model-based testing) rely on first obtaining a model of the system under analysis. System modeling is often done manually, which is often considered as a hindrance to adopt model-based system analysis and development techniques. To overcome this problem, researchers have proposed to automatically "learn" models based on sample system executions and shown that the learned models can be useful sometimes. There are however many questions to be answered. For instance, how much shall we generalize from the observed samples and how fast would learning converge? Or, would the analysis result based on the learned model be more accurate than the estimation we could have obtained by sampling many system executions within the same amount of time? Moreover, how well does learning scale to real-world applications? If the answer is negative, what are the potential methods to improve the efficiency of learning? In this work, we first investigate existing algorithms for learning probabilistic models for model checking and propose an evolution-based approach for better controlling the degree of generalization. Then, we present existing approaches to learn abstract models to improve the efficiency of learning for scalability reasons. Lastly, we conduct an empirical study in order to answer the above questions. Our findings include that the effectiveness of learning may sometimes be limited and it is worth investigating how abstraction should be done properly in order to learn abstract models.

Keywords Probabilistic model checking · Model learning · Genetic algorithm · Abstraction

1 Introduction

Many system analysis techniques rely on first obtaining a system model. The model should be accurate and often is required to be at a 'proper' level of abstraction. For instance, model checking [3,11] works effectively if the user-provided model captures all the relevant behavior of the system and abstracts away the irrelevant details. With such a model as well as a given property, a model checker would automatically verify the property or falsify it with a counterexample. Alternatively, in the setting of probabilistic model checking (PMC, see Sect. 2) [3,5], the model checker would calculate the probability of satisfying the property.

Model checking is perhaps not as popular as it ought to be due to the fact that a good model is required beforehand. For instance, a model which is too general would introduce spurious counterexamples, whereas the model checking result based on a model which under-approximates the relevant system behavior is untrustworthy. In the setting of PMC, users are required to provide a probabilistic model (e.g., a Markov chain [3]) with accurate probabilistic distributions, which is even more challenging.

In practice, system modeling is often done manually, which is both time-consuming and error-prone. What is worse, it could be infeasible if the system is a black box or it is so complicated that no accurate model is known (e.g., the chemical reaction in a water treatment system [49]). This is often considered by industry as one hindrance to adopt otherwise powerful techniques like model checking. Alternative approaches which would rely less on manual modeling have been explored in different settings. One example is

This research is partly supported by T2MOE1704 Singapore. Q. Yuan was supported by the National Research Fund (FNR), Luxembourg (Grant 7814267). J. Pang was partially supported by the project SEC-PBN (funded by the University of Luxembourg) and the ANR-FNR project AlgoReCell (INTER/ANR/15/11191283).

✉ Jingyi Wang
wangjyee@gmail.com

[1] Singapore University of Technology and Design, Singapore, Singapore

[2] University of Luxembourg, Luxembourg City, Luxembourg

Published online: 25 April 2018

statistical model checking (SMC, see Sect. 2) [47,62]. The main idea is to provide a statistical measure on the likelihood of satisfying a property, by observing sample system executions and applying standard techniques like hypothesis testing [4,21,62]. SMC is considered useful partly because it can be applied to black box or complex systems when system models are not available.

Another approach for avoiding manual modeling is to automatically learn models. A variety of learning algorithms have been proposed to learn a variety of models, e.g., [7,13,45,46]. It has been shown that the learned models can be useful for subsequent system analysis in certain settings, especially so when having a model is a must. Recently, the idea of model learning has been extended to system analysis through model checking. In [9,34,35], it is proposed to learn a probabilistic model first and then apply techniques like PMC to calculate the probability of satisfying a property based on the learned model. On the one hand, learning is beneficial, and it solves some known drawbacks of SMC or even simulation-based system analysis methods in general. For instance, since SMC relies on sampling *finite* system executions, it is challenging to verify unbounded properties [43,60], whereas we can verify unbounded properties based on the learned model through PMC. Furthermore, the learned model can be used to facilitate other system analysis tasks like model-based testing and software simulation for complicated systems. On the other hand, learning essentially is a way of generalizing the sample executions. It is thus worth investigating how the sample executions are generalized and whether indeed such learning-based approaches are justified.

In particular, we would like to investigate the following research questions. Firstly, how can we control the degree of generalization for the best learning outcome, since it is known that both over-fitting or under-fitting would cause problems in subsequent analysis? Secondly, often it is promised that the learned model would converge to an accurate model of the original system, if the number of sample executions is sufficiently large. In practice, there could be only a limited number of sample executions and thus it is valid to question how fast the learning algorithms converge. Furthermore, do learning-based approaches offer better analysis results if alternative approaches which do not require a learned model, like SMC, are available? Besides, how well does learning scale to real-world complex systems with many system variables? If not, do we have any approach to handle the issue?

Contributions We mainly make the following contributions in order to answer the above research questions. *Firstly, we propose a new approach* (Sect. 4) *to better control the degree of generalization than existing approaches* (Sect. 3) *in probabilistic model learning.* The approach is inspired by our observations on the limitations of existing learning approaches. Experiment results show that our approach converges faster and learns models which are much smaller than those learned by existing approaches while providing better or similar analysis results. We consider it is an advantage to learn smaller models as they are often easier to comprehensive and easier to model check. *Secondly, we develop a software toolkit* Ziqian, *realizing previously proposed learning approaches for PMC as well as our approach so as to systematically study and compare them in a fair way.* The tool is written in a generic manner and is friendly to extension. *Thirdly, we conduct an empirical study on comparing different model learning approaches against a suite of benchmark systems, two real-world systems, as well as randomly generated models* (Sect. 6). One of our findings suggests that learning models for model checking might not be as effective compared to SMC given the same time limit. However, the learned models may be useful when manual modeling is impossible. *Lastly, we investigate learning in the context of abstraction to deal with the state space explosion problem and reduce the cost of learning.* We show that current abstraction technique for probabilistic model learning can only work for a limited class of properties. Thus, it is worth investigating how to do abstraction for learning properly. From a broader point of view, our work is a first step toward investigating the recent trend on adopting machine learning techniques to solve software engineering problems. We remark that there is an extensive amount of existing research on learning non-probabilistic models (e.g., [1]), which is often designed for different usage and is thus beyond the scope of this work. We review related work and conclude this paper in Sect. 7.

2 Preliminary

In this work, the probabilistic model that we focus on is discrete-time Markov chains (DTMC) [3]. The reason is that most existing learning algorithms generate DTMC, and it is still ongoing research on how to learn other probabilistic models like Markov Decision Processes (MDP) [6,9,34,35,46]. Furthermore, the learned DTMC is intended for probabilistic analysis using methods like PMC. In the following, we briefly introduce DTMC, PMC as well as SMC.

Markov chain A DTMC $\mathcal{D}$ is a triple tuple $(S, \imath_{init}, Tr)$, where S is a countable, nonempty set of states; $\imath_{init} : S \rightarrow [0, 1]$ is the initial distribution s.t. $\sum_{s \in S} \imath_{init}(s) = 1$; and $Tr : S \times S \rightarrow [0, 1]$ is the transition probability assigned to every pair of states which satisfies the following condition: $\sum_{s' \in S} Tr(s, s') = 1$. $\mathcal{D}$ is *finite* if S is finite.

An example DTMC modeling the *egl* protocol [30,31] is shown in Fig. 1. The *egl* protocol is for exchanging commitment to a contract between parties who do not trust each other. Commitment is identified with the party's digital signature on the contract. The main property that the protocol is achieving is fairness such that in any case party A can obtain party B's commitment if party B already has part A's com-

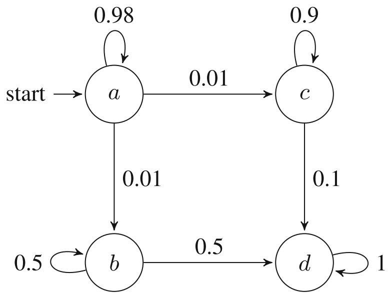

Fig. 1 DTMC of *egl* protocol

mitment. Figure [1] has 4 states which represent whether party A has obtained party B's commitment and the other way around. For example, the initial state a represents 00 which means that neither A or B has obtained each other's commitment and the terminal state d represents 11 which means that both A or B has obtained each other's commitment.

Paths of DTMCs are maximal paths in the underlying digraph, defined as infinite state sequences $\pi = s_0 s_1 s_2 \cdots \in S^\omega$ such that $Tr(s_i, s_{i+1}) > 0$ for all $i \geq 0$. We write $Path^{\mathcal{D}}(s)$ to denote the set of all infinite paths of $\mathcal{D}$ starting from state s. The probability of exhibiting a path fragment $\pi = s_0 s_1 \cdots s_n$ is given by $Tr(s_0, s_1) \times Tr(s_1, s_2) \times \cdots Tr(s_{n-1}, s_n)$.

Probabilistic model checking PMC [3,5] is a formal analysis technique for stochastic systems including DTMC and MDP. Given a DTMC $\mathcal{D} = (S, \imath_{init}, Tr)$ and a set of propositions Σ, we can define a function $L : S \rightarrow \Sigma$ which assigns a valuation of the propositions in Σ to each state in S. Once each state is labeled, given a path in $Path^{\mathcal{D}}(s)$, we can obtain a corresponding sequence of propositions labeling the states.

Let $\Sigma^\star$ and Σ^ω be the set of all finite and infinite strings over Σ, respectively. A property of the DTMC can be specified in temporal logic. Without loss of generality, we focus on Linear Time Temporal logic (LTL) and probabilistic LTL in this work. An LTL formula φ over Σ is defined by the syntax:

$$\varphi ::= true \mid \sigma \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathbf{X}\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

where $\sigma \in \Sigma$ is a proposition; $\mathbf{X}$ is intuitively read as 'next' and $\mathbf{U}$ is read as 'until'. We remark commonly used temporal operators like $\mathbf{F}$ (which reads 'eventually') and $\mathbf{G}$ (which reads 'always') can be defined using the above syntax, e.g., $\mathbf{F}\varphi$ is defined as $true\mathbf{U}\varphi$. Given a string π in $\Sigma^\star$ or Σ^ω, we define whether π satisfies a given LTL formula φ in the standard way [3].

Given a path π of a DTMC, we write $\pi \models \varphi$ to denote that the sequence of propositions obtained from π satisfies φ and $\pi \not\models \varphi$ otherwise. Furthermore, a probabilistic LTL formula ϕ of the form $Pr_{\bowtie r}(\varphi)$ can be used to quantify the probability of a system satisfying the LTL formula φ, where $\bowtie \in \{\geq, \leq, =\}$ and $r \in [0, 1]$ is a probability threshold. A DTMC $\mathcal{D}$ satisfies $Pr_{\bowtie r}(\varphi)$ if and only if the accumulated probability of all paths obtained from the initial states of $\mathcal{D}$ which satisfy φ satisfies the condition $\bowtie r$. Given a DTMC $\mathcal{D}$ and a probabilistic LTL property $Pr_{\bowtie r}(\varphi)$, the PMC problem can be solved using methods like the automata-theoretic approach [3]. We skip the details of the approach and instead remark that the complexity of PMC is doubly exponential in the size of φ and polynomial in the size of $\mathcal{D}$.

Statistical model checking SMC is a Monte Carlo method to solve the probabilistic verification problem based on system simulations. Its biggest advantage is perhaps that it does not require the availability of system models [12]. The idea is to provide a statistical measure on the likelihood of satisfying φ based on the observations by applying techniques like hypothesis testing [4,21,62]. In the following, we introduce how SMC works and refer readers to [3,62] for details.

Intuitively, SMC works by sampling system behaviors randomly (according to certain underlying probabilistic distribution) and observing how often a given property φ is satisfied. We then infer statistical property of the actual probability of the system satisfying φ. Because system simulations are finite, in order to tell whether a simulation satisfies φ, SMC is often limited to bounded properties, i.e., properties which can be validated or invalidated after a bounded number of steps.[1] Without loss of generality, we further restrict the property to be of the form: $\mathcal{P}_{\geq p}(\phi\, \mathcal{U}^{\leq t}\, \psi)$, which reads: there is a probability no less than p such that ϕ is always satisfied until ψ is in t time steps is.

Given a system of which we can reliably sample its behavior according to its underlying probabilistic distribution, SMC verifies a given bounded property using methods like hypothesis testing [59]. Hypothesis testing is a statistical process to decide the truthfulness of two mutual exclusive statements, say H_0 and H_1. In the setting of SMC, H_0 is the hypothesis that $\mathcal{P}_{\geq p}(\phi\ \mathcal{U}^{\leq t}\ \psi)$ is satisfied, and H_1 is the alternative hypothesis (i.e., that it is not satisfied). Besides, two parameters are required from users. One is the targeted assurance level, denoted as θ, over the system, and the other is a parameter σ used to identify the indifference region. The indifference region refers to the region $(\theta - \sigma, \theta + \sigma)$, which is used to avoid exhaustive sampling and obtain the desired control over the precision [62]. The probability of accepting H_1 given that H_0 holds (i.e., false negative) is required to be at most α and the probability of accepting H_0 if H_1 holds (i.e., false positive) should be no more than β. In practice, the error bounds (i.e., α, β) and σ can be decided by how much testing resource is available as more resource is required for a smaller error bounds or a smaller indifference region.

There are two main acceptance sampling methods to decide when the hypothesis testing procedure can be stopped.

[1] Refer to [43,60] for work on SMC of unbounded properties.

One is fixed-size sampling test, which often results in a large number of tests [62]. The other one is sequential probability ratio test (SPRT), which yields a variable sample size. SPRT is faster than fix-sampling methods as the testing process ends as soon as a conclusion is made. The basic idea of SPRT is to calculate the probability ratio, after observing a test result and comparing with two stopping conditions [53]. If either of the conditions is satisfied, the testing stops and returns which hypothesis is accepted. Readers can refer to [62] for details.

3 Probabilistic model learning

Learning probabilistic models from sampled system executions for the purpose of PMC has been explored extensively recently [7,9,13,34,35,45,46]. In this section, we briefly present existing probabilistic model learning algorithms for two different settings.

3.1 Learn from multiple executions

In the setting that the system can be reset and restarted multiple times, a set of independent executions of the system can be collected as input for learning. Learning algorithms in this category make the following assumptions [34]. First, the underlying system can be modeled as a DTMC. Second, the sampled system executions are mutually independent. Third, the length of each simulation is independent.

Let Σ denote the alphabet of the system observations such that each letter $e \in \Sigma$ is an observation of the system state. A system execution is then a finite string over Σ. The input in this setting is a finite set of strings $\Pi \subseteq \Sigma^{\star}$. For any string $\pi \in \Sigma^{\star}$, let $\mathit{prefix}(\pi)$ be the set of all prefixes of π including the empty string $\langle\rangle$. Let $\mathit{prefix}(\Pi)$ be the set of all prefixes of any string $\pi \in \Pi$. The set of strings Π can be naturally organized into a tree $\mathit{tree}(\Pi) = (N, \mathit{root}, E)$ where each node in N is a member of $\mathit{prefix}(\Pi)$; the root is the empty string $\langle\rangle$; and $E \subseteq N \times N$ is a set of edges such that (π, π') is in E if and only if there exists $e \in \Sigma$ such that $\pi \cdot \langle e\rangle = \pi'$ where $\cdot$ is the sequence concatenation operator.

The idea of the learning algorithms is to generalize $\mathit{tree}(\Pi)$ by merging the nodes according to certain criteria in certain fixed order. Intuitively, two nodes should be merged if they are likely to represent the same state in the underlying DTMC. Since we do not know the underlying DTMC, whether two states should be merged is decided through a procedure called *compatibility test*. We remark the compatibility test effectively controls the degree of generalization. Different types of compatibility test have been studied [7,29,44]. We present in detail the compatibility test adopted in the AALERGIA (hereafter AA) algorithm [34] as a representative. First, each node π in $\mathit{tree}(\Pi)$ is labeled with the number of strings *str* in Π such that π is a prefix of *str*. Let $L(\pi)$ denote its label. Two nodes π_1 and π_2 in $\mathit{tree}(\Pi)$ are considered compatible if and only if they satisfy two conditions. The first condition is $\mathit{last}(\pi_1) = \mathit{last}(\pi_2)$ where $\mathit{last}(\pi)$ is the last letter in a string π, i.e., if the two nodes are to be merged, they must agree on the last observation (of the system state). The second condition is that the future behaviors from π_1 and π_2 must be sufficiently similar (i.e., within Angluin's bound [2]). Formally, given a node π in $\mathit{tree}(\Pi)$, we can obtain a probabilistic distribution of the next observation by *normalizing* the labels of the node and its children. In particular, for any event $e \in \Sigma$, the probability of going from node π to $\pi \cdot \langle e\rangle$ is defined as: $Pr(\pi, \langle e\rangle) = \frac{L(\pi\cdot\langle e\rangle)}{L(\pi)}$. We remark the probability of going from node π to itself is $Pr(\pi, \langle\rangle) = 1 - \sum_{e\in\Sigma} Pr(\pi, \langle e\rangle)$, i.e., the probability of not making any more observation. The multi-step probability from node π to $\pi \cdot \pi'$ where $\pi' = \langle e_1, e_2, \ldots, e_k\rangle$, written as $Pr(\pi, \pi')$, is the product of the one-step probabilities:

$$Pr(\pi, \pi') = Pr(\pi, \langle e_1\rangle) \times Pr(\pi \cdot \langle e_1\rangle, \langle e_2\rangle) \\ \times \cdots \times Pr(\pi \cdot \langle e_1, e_2, \ldots, e_{k-1}\rangle, \langle e_k\rangle)$$

Two nodes π_1 and π_2 are compatible if the following is satisfied:

$$|Pr(\pi_1, \pi) - Pr(\pi_2, \pi)| < \sqrt{6\epsilon \log(L(\pi_1))/L(\pi_1)} \\ + \sqrt{6\epsilon \log(L(\pi_2))/L(\pi_2)}$$

for all $\pi \in \Sigma^{\star}$. We highlight that ϵ used in the above condition is a parameter which effectively controls the degree of state merging. Intuitively, a larger ϵ leads to more state merging, thus fewer states in the learned model.

If π_1 and π_2 are compatible, the two nodes are merged, i.e., the tree is transformed such that the incoming edge of π_2 is directed to π_1. Next, for any $\pi \in \Sigma^*$, $L(\pi_1 \cdot \pi)$ is incremented by $L(\pi_2 \cdot \pi)$. The algorithm works by iteratively identifying nodes which are compatible and merging them until there are no more compatible nodes. After merging all compatible nodes, the last phase of the learning algorithms normalizes the tree so that it becomes a DTMC.

Example Assume that we are given a set of 917 samples Π of the *egl* protocol (shown in Fig. 1) and construct the tree $\mathit{tree}(\Pi)$ accordingly as shown in Fig. 2. The labels on the nodes are the *numbers* of times the corresponding string is a prefix of some samples in Π. For instance, node a is labeled with 917 because $\langle a\rangle$ is the prefix of all samples (since it is the only initial state).

The tree can be viewed as the initial learned model which has no generalization. Next, the tree is generalized by merging nodes. Assume that node $\langle aa\rangle$ and node $\langle a\rangle$ in Fig. 2 pass compatibility test and are to be merged. Firstly, transitions to $\langle aa\rangle$ are directed to $\langle a\rangle$, which increases the number associated with $\langle a\rangle$ to 1817; then numbers labeled with decedents

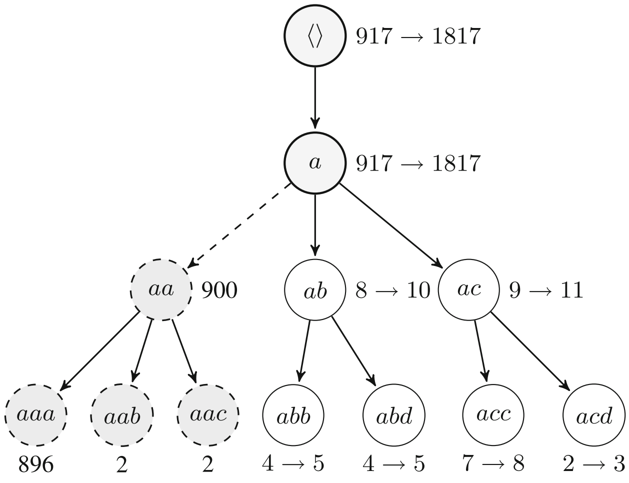

Fig. 2 Example tree representation of samples

of $\langle aa\rangle$ are added to the corresponding decedent nodes of $\langle a\rangle$. That is, the *numbers* of $\langle a\rangle$'s decedents are updated to the numbers after arrow as shown in Fig. 2. For instance, since the state label of $L(\langle aa\rangle\cdot\langle b\rangle)$ is 2, we update the label for node $\langle ab\rangle$ from 8 to 10. Afterward, the subtree rooted at $\langle aa\rangle$ (dashed circle nodes, aa inclusive) is pruned.

3.2 Learn from a single execution

In the setting that the system cannot be easily restarted, e.g., real-world cyber-physical systems. We are limited to observe the system for a long time and collect a single, long execution as input. Thus, the goal is to learn a model describing the long-run, stationary behavior of a system, in which system behaviors are decided by their finite variable length memory of the past behaviors.

In the following, we fix α to be the single system execution. Given a string $\pi = \langle e_0, e_1, \ldots, e_k\rangle$, we write *suffix*$(\pi)$ to be the set of all suffixes of π, i.e.,

$$\mathit{suffix}(\pi) = \{\langle e_i, \ldots, e_k\rangle | 0 \le i \le k\} \cup \{\langle\rangle\}.$$

Learning algorithms in this category [9,45] similarly construct a tree $\mathit{tree}(\alpha) = (N, \mathit{root}, E)$ where N is the set of suffixes of α; $\mathit{root} = \langle\rangle$; and there is an edge $(\pi_1, \pi_2) \in E$ if and only if $\pi_2 = \langle e\rangle\cdot\pi_1$. For any string π, let $\#(\pi, \alpha)$ be the number of times π appears as a substring in α. A node π in $\mathit{tree}(\alpha)$ is associated with a function Pr_π such that $Pr_\pi(e) = \frac{\#(\pi\cdot\langle e\rangle,\alpha)}{\#(\pi,\alpha)}$ for every $e \in \Sigma$, which is the likelihood of observing e next given the previous observations π. Effectively, function Pr_π defines a probabilistic distribution of the next observation. An example tree T is shown in Fig. 3. For simplicity, assume there are only two observations a and b. The numbers associated with the nodes are the predicted probability of having a and b (in this order) as the next observation.

Based on different suffixes of the execution, different probabilistic distributions of the next observation will be formed. For instance, the probabilistic distribution from the node $\langle\rangle$ would predict the distribution without looking at the history, whereas the node corresponding to the sequence of all previous observations would have a prediction based the entire history. The central question is how far we should look into the past in order to predict the future. As we observe more history, we will make a better prediction of the next observation. Nonetheless, constructing the tree completely (no generalization) is infeasible and the goal of the learning algorithms is thus to grow a part of the tree which would give a "good enough" prediction by looking at a small amount of history. The questions are then: what is considered "good enough" and how much history is necessary. The answers control the degree of generalization in the learned model.

In the following, we present the approach in [9] as a representative of algorithms proposed in the setting. Let $\mathit{fre}(\pi, \alpha) = \frac{\#(\pi,\alpha)}{|\alpha|-|\pi|-1}$ be the relative frequency of having substring π in α (where $|\pi|$ is the length of π). Algorithm 1 shows the algorithm for identifying the right tree by growing it on-the-fly. Initially, at line 1, the tree T contains only the root $\langle\rangle$. Given a threshold ϵ, we identify the set $S = \{\pi | \mathit{fre}(\pi, \alpha) > \epsilon\}$ at line 2, which are substrings appearing often enough in α and are candidate nodes to grow in the tree. The loop from line 3 to 7 keeps growing T. In particular, given a candidate node π, we find the longest suffix π' in T at line 4 and if we find that adding π would improve the prediction of the next observations by at least ϵ, π is added, along with all of its suffixes if they are currently missing from the tree (so that we maintain all suffixes of all nodes in the tree all the time). Whether we add node π into tree T or not, we update the set of candidate S to include longer substrings of α at line 6. When Algorithm 1 terminates, the tree contains all nodes which would make a *good enough* prediction. Afterward, the tree is transformed into a DTMC where the leafs of *tree*(α) are kept as states in the DTMC. We briefly introduce the transformation here and readers are referred to Appendix B of [45] for more details. For a state s and next symbol σ, the next state $s' = Tr(s, \sigma)$ is a suffix of $s\sigma$. However, this is not guaranteed to be a leaf in the learned T. Thus, the first step is to extend T to T' such that for every leaf s, the longest *prefix* of s is either a leaf or an internal node in T'. The transition functions are defined as follows. For each node s in $T \cap T'$ and $\sigma \in \Sigma$, let $P'(s, \sigma) = P(s, \sigma)$. For each new nodes s' in $T' - T$, let $P'(s', \sigma) = P(s, \sigma)$, where s is deepest ancestor of s' in T.

Example The left of Fig. 3 shows an example PST after learning. The three leafs of the tree will be taken as states in the DTMC, i.e., aa, ba and b. The transitions are formed by suffix

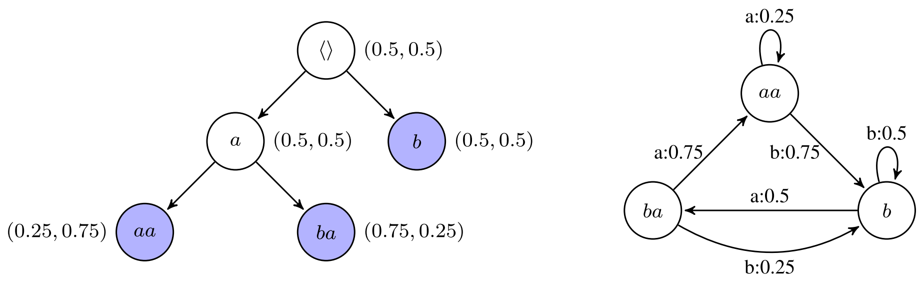

Fig. 3 The left figure is an example PST, where each node is associated with a distribution over all the symbols, i.e. $\{a, b\}$. The right figure is the DTMC model after transformation

Algorithm 1 *Learn PST*

```
Initialize $T$ to be a single root node representing $\langle\rangle$;
Let $S = \{\sigma | fre(\sigma, \alpha) > \epsilon\}$ be the candidate suffix set;
while $S$ is not empty do
    Take any $\pi$ from $S$; Let $\pi'$ be the longest suffix of $\pi$ in $T$;
    (B) If $fre(\pi, \alpha) \cdot \sum_{\sigma \in \Sigma} Pr(\pi, \sigma) \cdot \log \frac{Pr(\pi,\sigma)}{Pr(\pi',\sigma)} \geq \epsilon$
        add $\pi$ and all its suffixes which are not in $T$ to $T$;
    (C) If $fre(\pi, \alpha) > \epsilon$, add $\langle e \rangle \cdot \pi$ to $S$ for every $e \in \Sigma$ if $fre(\langle e \rangle \cdot \pi, \alpha) > 0$;
end while
```

matching. For example, starting from state ba, we go to state aa if we observe a because aa is a suffix of baa. Similarly, we go to state b if we observe b because b is the suffix of aab. Assume that the observation so far is $\alpha = \langle \cdots aba \rangle$. Given the model shown in Fig. 3, the next observations are predicted using the probability distribution of its longest suffix in the tree. For instance, the probability of observing a next would be predicted using the probability distribution associated with node $\langle ba \rangle$, which is $Pr_{\langle ba \rangle}(a) = 0.75$. For another example, the predicted probability to generate string $\langle abaa \rangle$ after α is computed as: $Pr_{\langle ba \rangle}(a) \cdot Pr_{\langle aa \rangle}(b) \cdot Pr_{\langle b \rangle}(a) \cdot Pr_{\langle ba \rangle}(a) = 0.75 \cdot 0.75 \cdot 0.5 \cdot 0.75$.

4 Learning through evolution

Model learning essentially works by generalizing the sample executions. The central question is thus how to control the degree of generalization. To find the best degree of generalization, both [34] and [9] proposed to select the 'optimal' ϵ value using the golden section search of the highest Bayesian Information Criterion (BIC) score. Golden section search is a technique which is suitable to find the minimum or maximum of a strictly unimodal function [58]. For instance, in [34], the BIC score of a learned model M, given the sample executions Π, is computed as follows: $log(Pr_M(\Pi)) - \mu \times |M| \times log(|\Pi|)$ where $|M|$ is the number of states in M; Π is the total number of observations and μ is a constant (set to be 0.5 in [34]) which controls the relative importance of the size of the learned model. This kind of approach to optimize BIC is based on the assumption that the BIC score is a concave function of the parameter ϵ. Our empirical study (refer to details in Sect. 6), however, shows that this assumption is flawed and the BIC score can fluctuate with ϵ.

In the following, we propose an alternative method for learning models based on genetic algorithms (GA) [26]. The method is designed to select the best degree of generalization without the assumption of BIC's concaveness. The idea is that instead of using a predefined ϵ value to control the degree of generalization, we systematically generate candidate models and select the ones using the principle of natural selection so that the "fittest" model is selected eventually. In the following, we first briefly introduce the relevant background on GA and then present our approach in detail.

4.1 Genetic algorithms

GA [26] are a set of optimization algorithms inspired by the "survival of the fittest" principle of Darwinian theory of natural selection. Given a specific problem whose solution can be encoded as a chromosome, a genetic algorithm typically works in the following steps [15]. First, an initial population (i.e., candidate solutions) is created either randomly or hand-picked based on certain criteria. Second, each candidate is evaluated using a predefined fitness function to see how good it is. Third, those candidates with higher fitness scores are selected as the parents of the next generation. Fourth, a new generation is generated by genetic operators, which either randomly alter (a.k.a. mutation) or combine fragments of their parent candidates (a.k.a. crossover). Lastly, step 2-4 are repeated until a satisfactory solution is found or some other termination condition (e.g., timeout) is satisfied. GA are especially useful in providing approximate 'optimal' solutions when other optimization techniques do not apply or are too expensive, or the problem space is too large or complex.

GA are suitable for solving our problem of learning DTMC because we view the problem as finding an optimal DTMC model which not only maximizes the likelihood of the observed system executions but also satisfies additional constrains like having a small number of states. To apply GA to solve our problem, we need to develop a way of encoding candidate models in the form of chromosomes, define operators such as mutation and crossover to generate new candidate models, and define the fitness function to selection better models. In the following, we present the details of the steps in our approach.

4.2 Learn from multiple executions

We first consider the setting where multiple system executions are available. Recall that in this setting, we are given a set of strings Π, from which we can build a tree representation $tree(\Pi)$. Furthermore, a model is learned through merging the nodes in $tree(\Pi)$. The space of different ways of merging the nodes thus corresponds to the potential models to learn. Our goal is to apply GA to search for the best model in this space. In the following, we first show how to encode different ways of merging the nodes as chromosomes.

Let the size of $tree(\Pi)$ (i.e., the number of nodes) be X and let Z be the number of states in the learned model. A way of merging the nodes is a function which maps each node in $tree(\Pi)$ to a state in the learned model. That is, it can be encoded as a chromosome in the form of a sequence of integers $\langle I_1, I_2, \ldots, I_X\rangle$ where $1 \leq I_i \leq Z$ for all i such that $1 \leq i \leq X$. Intuitively, the number I_i means that node i in $tree(\Pi)$ is mapped into state I_i in the learned model. Besides, the encoding is done such that infeasible models are always avoided. Recall that two nodes π_1 and π_2 can be merged only if $last(\pi_1) = last(\pi_2)$, which means that two nodes with different last observation should not be mapped into the same state in the learned model. Thus, we first partition the nodes into $|\Sigma|$ groups so that all nodes sharing the same last observation are mapped to the same group of integers. A chromosome is then generated such that only nodes in the same group can possibly be mapped into the same state. The initial population is generated by randomly generating a set of chromosomes this way. *We remark that in this way all generated chromosomes represent a valid DTMC model.*

Formally, the chromosome $\langle I_1, I_2, \ldots, I_X\rangle$ represents a DTMC $M = (S, \iota_{init}, Tr)$ where S is a set of Z states. Each state s in S corresponds to a set of nodes in $tree(\Pi)$. Let $nodes(s)$ denote that set. Tr is defined such that for all states s and s' in M,

$$Tr(s, s') = \frac{\sum_{x \in nodes(s)} \sum_{e \in \Sigma | \langle s,e\rangle \in nodes(s')} L(x \cdot \langle e\rangle)}{\sum_{x \in nodes(s)} L(x)},$$

where the nominator is the total number of occurrence of all the nodes grouped to state s and the denominator is the total number that we observe a transition from a node in state s to a node in state s'. Specifically, the initial distributions ι_{init} is defined such that for any state $s \in S$, $\iota_{init}(s) = \sum_{x \in nodes(s)} L(x)/L(\langle\rangle)$, where the nominator is the number of occurrence of node $\langle\rangle$ and the denominator is the total number that we observe a node in s right after $\langle\rangle$.[2]

Algorithm 2 Model learning by GA from multiple executions

```
Require: tree(5) and the alphabet $\Sigma$
Ensure: A chromosome encoding a DTMC $\mathcal{D}$
Let $Z$ be $|\Sigma|$; Let $Best$ be $null$;
repeat
    Let $population$ be an initial population with $Z$ states;
    Let $generation$ be 1;
    repeat
        Let $newBest$ be the fittest in $population$;
        if $newBest$ is fitter than $Best$ then
            Set $Best$ to be $newBest$;
        end if
        for all fit pairs $(p_1, p_2)$ in $population$ do
            Crossover $(p_1, p_2)$ to get children $C_1$ and $C_2$;
            Mutate $C_1$ and $C_2$;
            Add $C_1$ and $C_2$ into $population$;
            Remove $(p_1, p_2)$ from $population$;
        end for
        Select chromosomes with better fitness scores
        $generation \leftarrow generation + 1$;
    until $generation > someThreshold$
    $Z \leftarrow Z + 1$;
until $Best$ is not improved
return $Best$
```

Next, we define the fitness function. Intuitively, a chromosome is good if the corresponding DTMC model M maximizes the probability of the observed sample executions and the number of states in M is small. We thus define the fitness function of a chromosome as: $log(Pr_M(\Pi)) - \mu \times |M| \times log|\Pi|$ where $|M|$ is the number of states in M and $|\Pi|$ is the total number of letters in the observations and μ is a constant which represents how much we favor a smaller model size. The fitness function, in particular, the value of μ, controls the degree of generalization. If μ is 0, $tree(\Pi)$ would be the resultant model; whereas if μ is infinity, a model with one state would be generated. We remark that this fitness function is the same as the formula for computing the BIC score in [34]. Compared to existing learning algorithms, controlling the degree of generalization in our approach is more intuitive (i.e., different value of μ has a direct effect on the learned model). In particular, a single parameter μ is used in our approach, whereas in existing algorithms [9,34], a parameter μ is used to select the value of ϵ (based on a false assumption of the BIC being concave), which in turn controls the degree of generalization. From a user point of view, it is hard to see the effect of having a different ϵ value since it controls whether two nodes are merged in the intermediate steps of the learning process.

[2] Notice that if we consider node $\langle\rangle$ into consideration, the initial distribution will be 1 over $\langle\rangle$ since $\langle\rangle$ is not compatible with any other nodes.

Next, we discuss how candidate models with better fitness score are selected for the next round of evolution. Selection directs evolution toward better models by keeping good chromosomes and weeding out bad ones based on their fitness. Two standard selection strategies are applied. One is *roulette wheel selection.* Suppose f is the average fitness of a population. For each individual M in the population, we select f_M/f copies of M. The other is *tournament selection.* Two individuals are chosen randomly from the population and a tournament is staged to determine which one gets selected. The tournament is done by generating a random number r between zero and 1, and comparing it to a predefined number p (which is larger than 0.5). If r is smaller than p, the individual with a higher fitness score is kept. We refer the readers to [26] for discussion on the effectiveness of these selection strategies.

After selection, genetic operators like mutation and crossover are applied to the selected candidates. Mutation works by mapping a random node to a new number from the same group, i.e., merging the node with other nodes with the same last observation. For crossover, chromosomes in the current generation are randomly paired and two children are generated to replace them. Following standard approaches [26], we adopt three crossover strategies.

- *One-point crossover* A crossover point is randomly chosen, one child gets its prefix from the father and suffix from the mother. Reversely for the other child.
- *Two-point crossover* Two crossover points are randomly chosen, which results in two crossover segments in the parent chromosomes. The parents exchange their crossover segments to generate two children.
- *Uniform crossover* One child gets its odd bit from father and even bit from mother. Reversely for the other child.

We remark that during mutation or crossover, we guarantee that only chromosomes representing valid DTMC models are generated, i.e., only two nodes with the same last observations are mapped to the same number (i.e., a state in the learned model).

The details of our GA-based algorithm is shown as Algorithm 2. Variable Z is the number of states in the learned model. We remark that the number of states in the learned model M is unknown in advance. However, it is at least the number of letters in alphabet Σ, i.e., when all nodes in *tree(5)* sharing the same last observation are merged. Since a smaller model is often preferred, the initial population is generated such that each of the candidate models is of size $|\Sigma|$. The size of the model is incremented by 1 after each round of evolution. Variable $Best$ records the fittest chromosome generated so far, which is initially set to be *null* (i.e., the least fit one). At line 3, an initial population of chromosome with Z states are generated as discussed above. The loop from line 5 to 18 then lets the population evolve through a number of generations, during which crossover, mutations and selection take place. At line 19, we then increase the number of states in the model in order to see whether we can generate a fitter chromosome. We stop the loop from line 2 to 20 when the best chromosome is not improved after increasing the number of states. Lastly, the fittest chromosome $Best$ is decoded to a DTMC and presented as the learned model.

Table 1 GA encoding with 4 states

a	aa	ab	aac	abd	acd	$aacd$
1	1	2	3	4	4	4

Table 2 GA encoding with 5 states

a	aa	ab	aac	abd	acd	$aacd$
1	1	2	3	4	4	5

Example We use an example to illustrate how the above approach works. For simplicity, assume we have the following collection of executions $\Pi = \{\langle aacd\rangle, \langle abd\rangle, \langle acd\rangle\}$ from the model shown in Fig. 1. There are in total 7 prefixes of these execution (including the empty string). As a result, the tree $tree(\Pi)$ contains 8 nodes. Since the alphabet $\{a, b, c, d\}$ has size 4, the nodes (except the root) are partitioned into 4 groups so that all nodes in the same group have the same last observation. The initial population contains a single model with 4 states, where all nodes in the same groups are mapped into the same state as shown in Table 1. After one round of evolution, models with 5 states are generated (by essentially splitting the nodes in one group to two states as shown in Table 2, $aacd$ is split from abd and acd) and evaluated with the fitness function. The evolution continues until the fittest score does not improve anymore when we add more states.

4.3 Learn from a single execution

In the following, we describe our GA-based learning if there is only one system execution. Recall that we are given a single long system observation α in this setting. The goal is to identify the *shortest dependent history memory* that yields the most precise probability distribution of the system's next observation. That is, we aim to construct a part of $tree(\alpha)$ which transforms to a "good" DTMC. A model thus can be defined as an assignment of each node in $tree(\alpha)$ to either true or false. Intuitively, a node is assigned true if and only if it is selected to predict the next observation, i.e., the corresponding suffix is kept in the tree which later is used to construct the DTMC model. A chromosome (which encodes a model) is thus in the form of a sequence of boolean variable $\langle B_1, B_2, \ldots, B_m \rangle$ where B_i represents whether the i-th node is to be kept or not. We remark that not every valuation of the boolean variables is considered a valid chromosome. By definition, if a suffix π is selected to predict the next observation, all suffixes of π are not selected (since using a longer memory as in π predicts better) and therefore their corresponding value must be false. During mutation and crossover, we only generate those chromosomes satisfying this condition so that only valid chromosomes are generated.

A chromosome defined above encodes a part of $tree(\alpha)$, which can be transformed into a DTMC following the approach in [45]. Let M be the corresponding DTMC. The fitness function is defined similarly as in Sect. 4.2. We define the fitness function of a chromosome as $log(Pr_M(\alpha)) - \mu \times |M| \times log(|\alpha|)$ where $Pr_M(\alpha)$ is the probability of exhibiting α in M, μ is a constant that controls the weight of model size, and $|\alpha|$ is the size of the input execution. Mutation is done by randomly selecting one boolean variable from the chromosome and flip its value. Notice that afterward, we might have to flip the values of other boolean values so that the chromosome is valid. We skip the discussion on selection and crossover as they are the same as described in Sect. 4.2.

We remark that, compared to existing algorithms in learning models [9,34,35], it is straightforward to argue that the GA-based approaches for model learning do not rely on the assumption needed for BIC. Furthermore, the learned model improves monotonically through generations.

5 Learning based on abstraction

Probabilistic model learning is potentially very expensive. Imagine a system with many observable variables, the alphabet size for learning will be large. For instance, the PRISM model of *egl* protocol has dozens of integer or Boolean variables [30,31]. Even worse, if there exist real-typed (e.g. double or float) variables, the alphabet size will be infinite which immediately renders learning infeasible. One solution to this problem is to apply abstraction to system traces before learning and learn abstract system models based on the abstract traces instead. Figure 4 shows an overview of learning abstract models. Given the original system traces, we first project each concrete state into the abstract state space defined by some abstraction functions. Then, we apply the above-mentioned learning algorithms to learn abstract models from the abstract traces in a standard way. Thus, the central question is, how should we perform abstraction to obtain abstract traces?

In general, abstraction should be done at a 'proper' level. A too coarse abstraction would leave out too much information. The learned model afterward may not be precise enough to verify a given property. In contrast, the learning cost will keep high if the abstraction is too conservative and too many details are kept. In the following, we present two kinds of abstraction technique from literature and pose the remaining research challenges to identify a 'proper' level of abstraction in terms of verifying or falsifying a property.

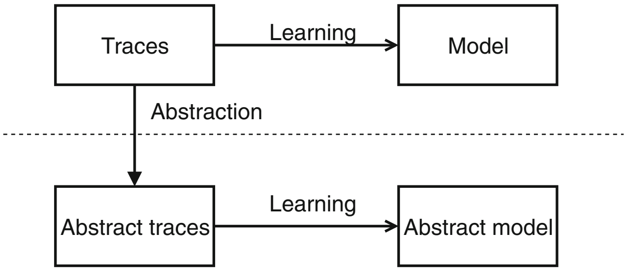

Fig. 4 An overview of learning an abstract model

5.1 Abstraction by filtering irrelevant variables

A direct approach is to take the properties to verify into account and abstract away all the variables which are irrelevant to the properties. Suppose φ is the property to verify, V is the set of all variables of the system and V_φ is the set of variables that appeared in φ. We abstract each system observation $e \in \Sigma$ by removing variables in $\{v | v \in V \text{ and } v \notin V_\varphi\}$. By doing so, we derive an abstract alphabet Σ_{V_φ}, where the size of the abstract alphabet is reduced to the combination of the variables relevant to the property only. We thus obtain the abstract system traces by abstract each system observation one by one. Afterward, we can apply a learning algorithm described in the above sections to learn the abstract model.

5.2 Predicate abstraction

Abstraction by filtering irrelevant variables as described above could improve the efficiency of learning in some cases. However, if there remain real-typed variables after the abstraction, the alphabet for learning will still be infinite, which renders learning infeasible. In the following, we present predicate abstraction, which is proposed in [16], to tackle the infinite state space problem for learning.

A predicate φ is a Boolean expression over the set of system variables V. Given a set of predicates $\{\varphi_1, \varphi_2, \ldots, \varphi_n\}$, we can map a system observation into an abstract system observation, which is a bit vector of length n and the i-th bit is 1 if φ_i is evaluated to be true at current system observation and 0 otherwise. Thus, the alphabet size after predicate abstraction is bounded by 2^n even when the original alphabet size is infinite. An example of predicate abstraction is given in Fig. 5. Similarly, we can obtain the abstract system traces by abstract each system observation one by one and learn the abstract model based on the abstract system traces afterward.

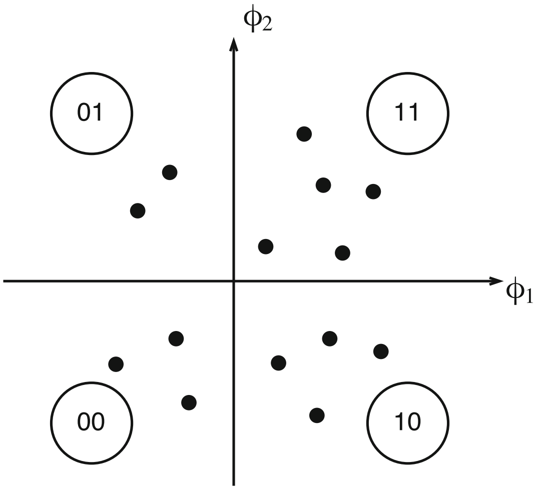

Fig. 5 An example of predicate abstraction given two predicates φ_1 and φ_2. A black dot is a concrete system observation. An abstract system observation 10 represents those concrete observations where φ_1 is true and φ_2 is false

5.3 A 'proper' level of abstraction

A selection of variables or the set of predicates defines an abstraction level over the system traces. The central question is how to choose a 'proper' set of variables or predicates which leads to a 'proper' level of abstraction. Intuitively, a 'proper' abstraction should just capture all the information relevant to the property and based on which we learn the abstract model. One basic heuristic is to take the property to verify into account and only take those variables or predicates that are relevant to the property as the abstraction strategies described above. It is proved in [41] that this kind of abstraction can only work under certain conditions. One condition is that the property φ we are verifying is the bounded fragment of PLTL. Or alternatively, the learning algorithm we are using converges for non-determinism model learning [14] as well. In these cases, we can guarantee the correctness of $Pr(M \models \varphi) = Pr(M^{\#} \models \varphi)$ in the limit ($M^{\#}$ is the learned abstract model). However, a deterministic learning algorithm like AA may not converge anymore after we adopt abstraction [41], since the abstraction introduces non-determinism in the trace level. As a result, we cannot reliably verify the property on the learned abstract model, which is problematic. Thus, *how to identify a 'proper' level of abstraction in general remains a research challenge.*

6 Empirical study

We implemented both state-of-the-art and GA-based learning algorithms from both multiple executions and a single execution together in a self-contained tool called ZIQIAN for systematic comparison. The tool and its usage is available at GitHub [54] with approximately 6K lines of Java code. The tool makes use of a parallel evolutionary computation engine called WatchMaker [15] for GA-based learning. ZIQIAN also supports learning at a user-defined abstraction level. Besides, it has recently been extended to support automatic abstraction refinement to verify or falsify a given safety property. More details could be found at [54] and [57].

In this work, since the primary goal of learning the models is to verify properties over the systems, we evaluate the learning algorithms by checking whether we can reliably verify properties based on the learned model, by comparing verification results based on the learned models and those based on the actual models (if available). All results are obtained using PRISM [32] on a 2.6 GHz Intel Core i7 PC running OSX with 8 GB memory. The constant μ in the fitness function of learning by GA is set to 0.5. *We acknowledge that the learned models could be useful in many other ways and it is beyond the scope of this work to evaluate whether they are useful in general.*

Our test objects can be categorized in two groups. The first group contains all systems (*brp*, *lse*, *egl*, *crowds*, *nand*, and *rsp*) from the PRISM benchmark suite for DTMCs [30] and a set of randomly generated DTMC models (*rmc*) using an approach similar to the approach in [50]. We refer the readers to [30,31] for details on the PRISM models as well as the properties to be verified. For these models, we collect multiple executions. The second group contains two real-world systems, from which we collect a single long execution. One is the probabilistic boolean networks (*PBN*), which is a modeling framework widely used to model gene regulatory networks (GRNs) [48]. In *PBN*, a gene is modeled with a binary valued node and the interactions between genes are expressed by Boolean functions. For the evaluation, we generate random PBNs with 5, 8 and 10 nodes, respectively, using the tool ASSA-PBN [38]. The other is a real-world raw water purification system called the Secure Water Testbed (*SWaT*) [49]. *SWaT* is a complicated system which involves a series of water treatments like ultra-filtration, chemical dosing, dechlorination through an ultraviolet system, etc. We regard *SWaT* as a representative complex system for which learning is the only way to construct a model. Our evaluation consists of the following parts (all models as well as the detailed results are available at [55]). We have the following findings by conducting the empirical study over the above test subjects.

Finding 1. Assumptions required by existing learning algorithms may not hold, which motivates our proposal of GA-based algorithms. Existing learning algorithms [9,34] require that the BIC score is a unimodal function of ϵ in order to select the best ϵ value which controls the degree of generalization. Figure 6 shows how the absolute value of

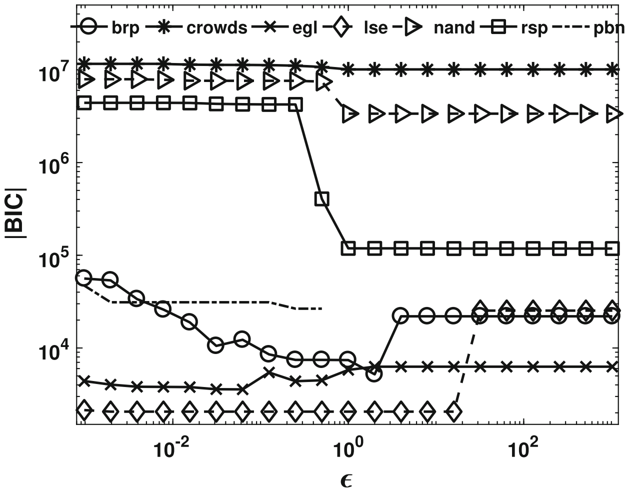

Fig. 6 How the absolute values of BIC score change over ϵ

BIC scores ($|BIC|$) of representative models change with ϵ. It can be observed that this assumption is not satisfied and ϵ is not controlling the degree of generalization nicely. For example, the $|BIC|$ (e.g., for *brp*, *PBN* and *egl*) fluctuate with ϵ. Besides, we observe climbings of $|BIC|$ for *lse* when ϵ increases, but droppings for *crowds*, *nand* and *rsp*. What's worse, in the case (e.g., *PBN*) of learning from a single execution, if the range of ϵ is selected improperly, it is very likely that an empty model (a tree only with root $\langle\rangle$) is learned.

Finding 2. Both GA and AA converge to more accurate results if sufficient time for learning is given. In the rest of the section, we adopt absolute relative difference (ARD) as a measure of accuracy of different approaches. The ARD is defined as $|P_{est} - P_{act}|/P_{act}$ between the precise result P_{act} and the estimated results P_{est}, which can be obtained by AA, GA as well as SMC. A smaller ARD implies a better estimation of the true probability. Figure 7 shows how the ARD of different systems change when we gradually increase the time cost from 30 s to 30 min by increasing the size of training data. We remark that some systems (*brp*, *egl*, *lse*) are not applicable due to different reasons explained later. In general, both AA and GA converge to relatively accurate results when we are given sufficient time. But there are also cases of fluctuation of ARD, which is problematic in reality, as in such cases, we would not know which result to trust (given the different verification results obtained with different number of sampled executions), and it is hard to decide whether we have gathered enough system executions for reliable verification results.

Finding 3. Statistical model checking always produces more accurate results, however, the learned model could be useful for later analysis. We compare the accuracy of AA, GA, and SMC for benchmark systems given the same amount of time in Fig. 8. We remark that due to the discrimination of system complexity (state space, variable number/type, etc.), different systems can converge in different speed. For SMC, we adopt the statistical model checking engine of PRISM and select the confidence interval method. We fix confidence to 0.001 and adjust the number of samples to adjust time cost. We have the following observations based on Fig. 8. Firstly, for most systems, GA results in more accurate results than AA given same amount of time. This is especially true if sufficient time (20m or 30m) are given. *However, it should be noticed that SMC produces significantly more accurate results*. Secondly, we observe that model learning works well if the actual model contains a small number of states. Cases like random models with 8 states (rmc-8) are good examples. For systems with more states, the verification results could deviate significantly (like nand-20-3, rsp-11).

Among our test subjects, *PBN* and *SWaT* are representative systems for which manual modeling is extremely challenging. Furthermore, SMC is not applicable as it is infeasible to sample the executions many times for these systems. We evaluate whether we can learn precise models in such a scenario.

For *PBN*, we use the tool ASSA-PBN [38] to generate the data to learn from. First, a set of steady states which represents whether a gene node is activated are generated. Notice

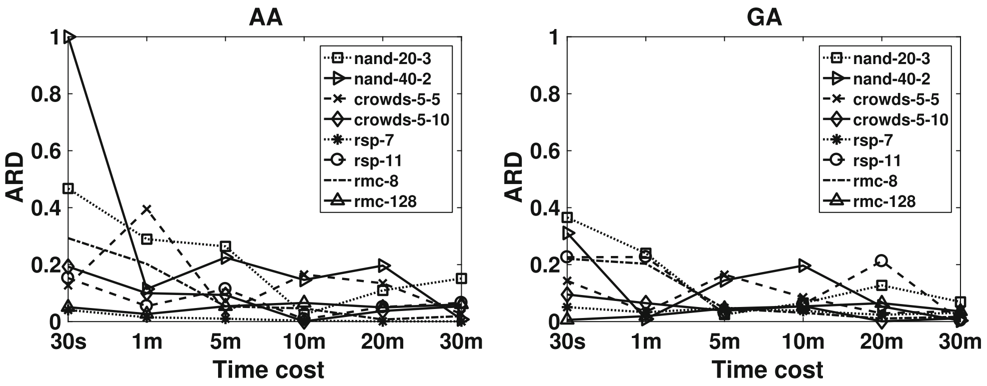

Fig. 7 Convergence of AA and GA over time. The numbers after the system of legends are one kind of system configuration

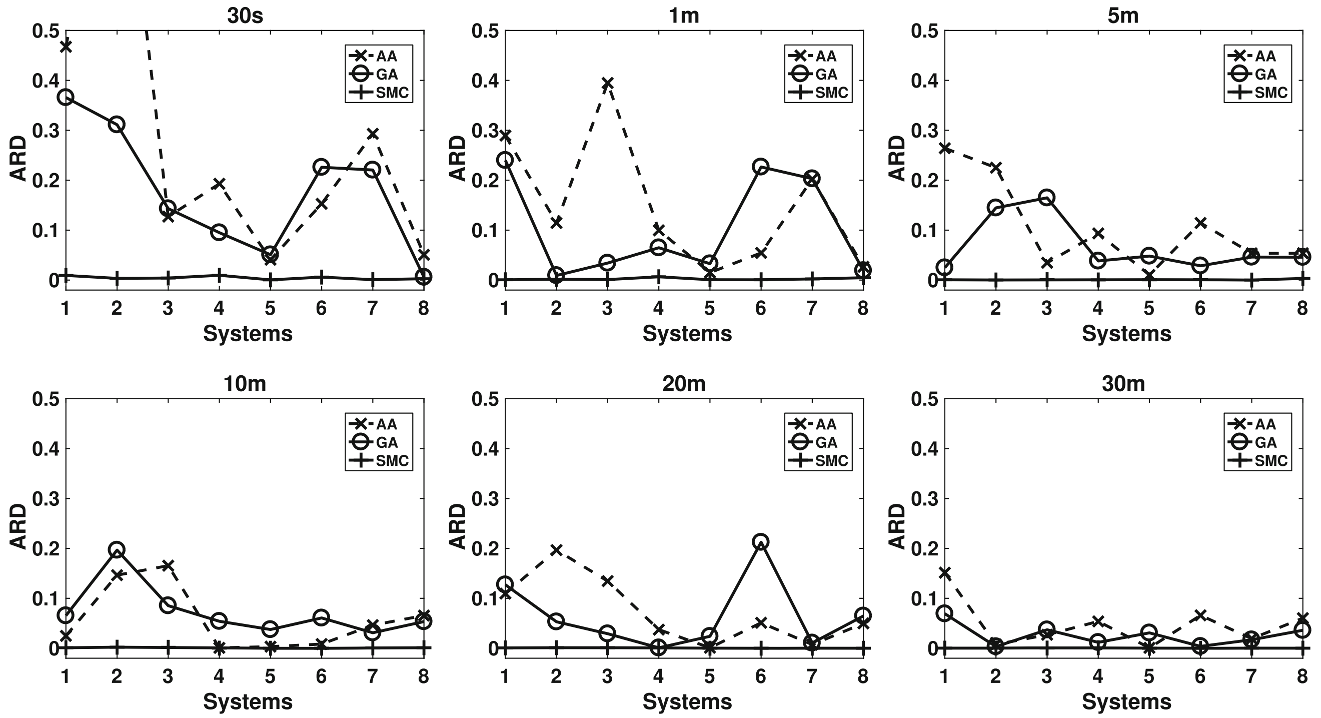

Fig. 8 The comparison of accuracy of AA, GA, and SMC given same amount of time, which varies from 30 s to 30 min. The horizontal-axis point represents a benchmark system with certain configuration in Fig. 7

that the number of states depends on the number of gene nodes. Then, transition probabilities are assigned between each state. In this way, we can evaluate the accuracy of the learned model directly with the generating model by comparing the differences between the transition probabilities. Following [48], we use mean squared error (MSE) to measure how precise the learned models are. MSE is computed as follows: $MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$ where n is the number of states in *PBN* and Y_i is the steady-state probabilities of the original model and $\hat{Y}_i$ is the corresponding steady-state probabilities of the learned model. We remark that the smaller its value is, the more precise the learned model is. Table 3 shows the MSE of the learned models with for *PBN* with 5, 8, and 10 nodes, respectively. Note that AA and GA learn the same models and thus have the same MSE, while GA always consumes less time. We can observe the MSEs are very small, which means the learned models of *PBN* are reasonably precise.

For the *SWaT* system, since it is a real-world system of which we do not have the actual model, we must define the preciseness of the learned model without referring to the actual model. We propose to evaluate the accuracy of the learned models by comparing the predicted observations against a set of test data collected from the actual system.

Table 3 Results of *PBN* steady-state learning

# Nodes	# States	Trajectory size ($\times 10^3$)	Time cost (s)		MSE ($\times 10^{-7}$)	# Nodes	# States	Trajectory size ($\times 10^3$)	Time cost (s)		MSE ($\times 10^{-7}$)
			PST	GA					PST	GA	
5	32	5	37.28	6.37	36.53	8	256	5	29.76	2.36	1.07
		15	161.57	53.49	15.21			15	105.87	26.4	0.03
		25	285.52	182.97	6.04			25	197.54	73.92	0.37
		35	426.26	348.5	7.75			35	310.87	122.61	0.94
		45	591.83	605.1	5.74			45	438.09	429.81	0.78
		50	673.55	767.7	4.28			50	509.59	285.66	0.34
10	1024	5	902.69	266.74	1.78	10	1024	15	5340.54	2132.68	0.61
		10	2772.56	1010.16	1.01			20	8477.24	3544.82	0.47

Table 4 Results of learned abstract model of *egl* protocol

System	Parameters	Property	Actual	SMC	AA	GA
egl	$L = 2, N = 5$	Unfair A	0.5156	0.505	0.4961	0.4961
		Unfair B	0.4844	0.472	0.5039	0.5039
	$L = 2, N = 10$	Unfair A	0.5005	0.525	0.4619	0.4619
		Unfair B	0.4995	0.494	0.5381	0.5381

The intuition behind is that the learned model is considered to be more precise if it could generate the testing data with high probability. Thus, we compare the average probability of generating each observation in the testing data referring to the learned model, which is defined as $\bar{P}_{obs} = P_{td}^{1/|td|}$, where td is the test data, $|td|$ is its length and P_{td} is the probability of generating td with the learned model, to evaluate how good the models are. The higher the probability, the more precise is the learned model considered to be. In particular, we apply steady-state learning proposed in [9] (hereafter PST) and GA to learn from executions of different length and observe the trends over time. We select 3 critical sensors in the system (out of 50), named $ait502$, $ait504$ and $pit501$, and learn models on how the sensor readings evolve over time (a more complete case study can be found in [28]). During the experiments, we find it very difficult to identify an appropriate ϵ for PST in order to learn a nonempty useable model. Our GA-based approach however does not have such problem. Eventually we managed to identify an optimal ϵ value and both PST and GA learn the same models given the same training data. A closer look at the learned models reveals that they are all first-order Markov chains. This makes sense in the way that sensor readings in the real *SWaT* system vary slowly and smoothly. Applying the learned models to predict the probability of the test data (from another day with length 7000), we observe a very good accuracy. In our experiment, the average generating probability using the learned model for $ait502$ and $pit501$ is over 0.97, and the number is 0.99 for $ait504$, which are reasonably precise.

Finding 4. GA learns models with much fewer states than AA. Models with fewer states are better than models with many states since they are easier for humans to understand the system. We thus compare the number of states learned by GA and AA, respectively. The results are shown in Table 5. It can be seen that GA usually learns models with significantly fewer states than models learned by AA. The reason is that GA always starts with a model with the number of states being the size of the learning alphabet. The model size increases only when adding a state will significantly improve our generalization of the system traces. On the other hand, the model learned by AA may contain a lot more states. The reasons are as follows. Note that every prefix node is potentially a system state in the learned model, which is much more than the alphabet size. AA compares the difference of future distributions of two prefix nodes and merges them if they are similar enough decided by parameter ϵ. A strict bound ϵ may lead to a model with even more states. Notice that for small models with few states, AA and GA may agree on the learned models.

Table 5 Comparison of number of states in the learned models by AA and GA respectively

Data size	nand-20-3		crowds-5-5		rsp-7		rmc-8	
	AA	GA	AA	GA	AA	GA	AA	GA
10,000	1717	88	5062	97	181	128	168	8
20,000	1749	90	8285	100	128	128	181	8
30,000	1714	90	10,232	100	128	128	44	8
40,000	1866	90	13,269	100	128	128	21	8
50,000	2040	90	16,190	101	128	128	24	8

Finding 5. Abstraction reduces the cost of learning significantly, however, it is worth investigating how abstraction should be done under different scenarios. Learning does not work when the state space of underlying system is too large or even infinite. If there are too many system variables to observe (or when float/double typed variables exist), which induces a very large (or even infinite) state space, learning will become infeasible. For example, to verify the fairness property of *egl* protocol, we need to observe dozens of integer variables. Our experiment suggests that AA and GA take unreasonable long time to learn a model, e.g., more than days. In order to apply learning in this scenario, we thus have to apply abstraction on the sampled system executions and learn from the abstract traces. Only by doing so, we are able to reduce the learning time significantly. In fact, to learn models in reasonably long time (e.g. within hours), we already manually select variables, i.e., filtering some irrelevant variables, for *nand* and *crowds* protocol. In the process, we find that if we abstract away all the relevant variables or else the set of variables is not selected properly, the verification results will always be 1 for unbound properties. Meanwhile, we also applied predication abstraction for *egl* protocol using the two predicates in the property to verify and successfully verified the properties. Notice that the properties to verify for *egl* protocol are unbounded. However, how to identify the 'proper' level of abstraction (select the right set of variables or predicates) is highly non-trivial in general and is to be investigated in the future (Table 4).

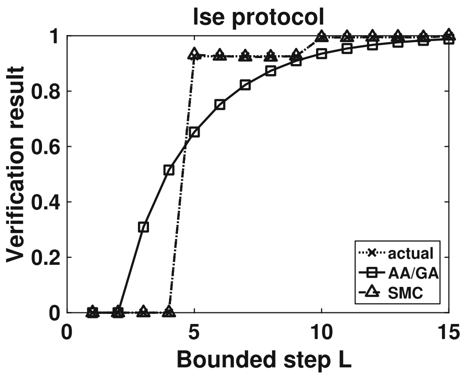

Fig. 9 Inconsistencies between the verification results based on the learned model and the actual results

Finding 6. Both learning and statistical model checking suffer from the rare event problem. The rare event problem is a major threat to the validity of both learning and statistical model checking. For the *brp* system, the probability of satisfying the given properties are very small. As a result, a system execution satisfying the property is unlikely to be observed and learned from. Consequently, the verification results based on the learned models are 0. It is confirmed that standard SMC is also ineffective for these properties since it is also based on random sampling. One possible solution to tackle rare event problem is to combine importance sampling with model learning, which we regard as future direction.

Finding 7. There are other complications which might make learning ineffective. For the *lse* protocol, the verification results based on the learned models may deviate from actual results for properties that show the probability of electing a leader in L rounds, with a different value for L. Figure 9 shows how the verification results change when we change the bounded step L. Notice that AA and GA learn exactly the same models. While the actual results 'jump' twice as L increases, the results based on the learned model are smooth and deviate from actual results significantly when L is 3, 4 or 5, while results based on SMC are consistent with the actual results. One possible reason is that there are some unfortunate state merging, which merges the states which will never satisfy the property to those states who will satisfy the property.

7 Related work

This work is initially inspired by the recent work on adopting machine learning to learn a variety of system models (e.g., DTMC, stationary models and MDPs) for model checking in order to avoid manual model construction [9,34–37]. This work is an attempt to empirically study whether such kind of learning approaches are applicable in real-world settings. In [41], an abstract model is learned for statistical model checking. There is also a recent effort which aims to build an abstract probabilistic model to verify safety PLTL properties [57]. They start with the coarsest abstraction and iteratively add more details (in the form of a new predicate) to refine the abstraction until a property is verified or falsified.

The study of such learning algorithms is often based on grammar inference [13], which traces back to automata learning [1,51]. Existing probabilistic model learning algorithms are often based on algorithms designed for learning deterministic (probabilistic) finite automata, which are investigated and evidenced in many previous works including but not limited to [7,8,10,18–20,25,44,45,52]. It is also related to the work on Markov chain estimation [17,56].

SMC [12,33,47,62,63] is the main competitor of learning-based approaches for model checking when a system model is not available. There are some recent work on extending SMC to unbounded properties [43,60]. Besides, our proposal on adopting genetic algorithms is related to work on applications of evolutionary algorithms for system analysis. In [22], genetic algorithm is integrated to abstraction refinement for model checking.

This work is also remotely related to the work in [46], which learns continuous time Markov chains. In addition, in [6], learning algorithms are applied in order to verify Markov decision processes, without constructing explicit models. The work is also remotely related to the previous work comparing the effectiveness of PMC and SMC [61]. Lastly, this work relies on the PRSIM model checker as the verification engine [32] and the case studies are taken from various practical systems and protocols including [23,24,27, 38–40,42].

8 Conclusion

In this work, we investigate the validity of probabilistic model learning for the purpose of probabilistic model checking. We also propose a novel GA-based approach to overcome limitations of existing probabilistic model learning algorithms. To reduce learning cost and make learning more realistic, we introduce two kinds of abstraction techniques for learning abstract models. Lastly, we conducted an empirical study to systematically evaluate the effectiveness and efficiency of all these probabilistic model learning approaches compared to statistical model checking over a variety of systems. We also discuss the potential challenges to adopt probabilistic model learning for model checking to real-life applications and introduce a possible direction to solve the problem.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)

2. Angluin, D.: Identifying languages from stochastic examples (1988). Technical Report YALEU/DCS/RR-614 (Yale University, Department of Computer Science)
3. Baier, C., Katoen, J.-P., et al.: Principles of Model Checking, vol. 26202649. MIT Press, Cambridge (2008)
4. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, pp. 260–272. Springer (2006)
5. Bianco, A., De Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Foundations of Software Technology and Theoretical Computer Science, pp. 499–513. Springer (1995)
6. Brázdil, T., Chatterjee, K., Chmelík, M., Forejt, V., Křetínskỳ, J., Kwiatkowska, M., Parker, D., Ujma, M.: Verification of Markov decision processes using learning algorithms. In: Automated Technology for Verification and Analysis, pp. 98–114. Springer (2014)
7. Carrasco, R.C., Oncina, J.: Learning stochastic regular grammars by means of a state merging method. In: Grammatical Inference and Applications, pp. 139–152. Springer (1994)
8. Carrasco, R.C., Oncina, J.: Learning deterministic regular grammars from stochastic samples in polynomial time. Inf. Theor. Appl. **33**(1), 1–19 (1999)
9. Chen, Y., Mao, H., Jaeger, M., Nielsen, T.D., Larsen, K.G., Nielsen, B.: Learning Markov models for stationary system behaviors. In: NASA Formal Methods, pp. 216–230. Springer (2012)
10. Clark, A., Thollard, F.: Pac-learnability of probabilistic deterministic finite state automata. J. Mach. Learn. Res. **5**, 473–497 (2004)
11. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
12. Clarke, E.M., Zuliani, P.: Statistical model checking for cyber-physical systems. In: Automated Technology for Verification and Analysis, pp. 1–12. Springer (2011)
13. De la Higuera, C.: Grammatical Inference, vol. 96. Cambridge University Press, Cambridge (2010)
14. Denis, F., Esposito, Y., Habrard, A.: Learning rational stochastic languages. In: COLT, vol. 4005, pp. 274–288. Springer (2006)
15. Dyer, D.W.: Watchmaker framework for evolutionary computation. http://watchmaker.uncommons.org. Accessed 23 Apr 2018
16. Graf, S., Saïdi, H.: Construction of abstract state graphs with pvs. In: International Conference on Computer Aided Verification, pp. 72–83. Springer (1997)
17. Guédon, Y.: Estimating hidden semi-Markov chains from discrete sequences. J. Comput. Graph. Stat. **12**(3), 604–639 (2003)
18. Guttman, O., Vishwanathan, S.V.N., Williamson, R.C.: Learnability of probabilistic automata via oracles. In: International Conference on Algorithmic Learning Theory, pp. 171–182. Springer (2005)
19. Habrard, A., Bernard, M., Sebban, M.: Improvement of the state merging rule on noisy data in probabilistic grammatical inference. In: European Conference on Machine Learning, pp. 169–180. Springer (2003)
20. Hammerschmidt, C.A., Verwer, S., Lin, Q., State, R.: Interpreting finite automata for sequential data. arXiv preprint arXiv:1611.07100 (2016)
21. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 342–356. Springer (2002)
22. He, F., Song, X., Hung, W.N.N., Gu, M., Sun, J.: Integrating evolutionary computation with abstraction refinement for model checking. IEEE Trans. Comput. **59**(1), 116–126 (2010)
23. Leen, H., Sellink, M.P.A., Vaandrager, F.W.: Proof-Checking a Data Link Protocol. Springer, Berlin (1994)
24. Herman, T.: Probabilistic self-stabilization. Inf. Process. Lett. **35**(2), 63–67 (1990)
25. Heule, M.J.H., Verwer, S.: Exact dfa identification using sat solvers. In: International Colloquium on Grammatical Inference, pp. 66–79. Springer (2010)
26. Holland, J.H.: Adaptation in Natural and Artificial Systems. MIT Press, Cambridge (1992)
27. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Inf. Comput. **88**(1), 60–87 (1990)
28. Wang, J., Sun, J., Jia, Y., Qin, S., Xu, Z.: Toward 'verifying' a water treatment system. arXiv preprint arXiv:1712.04155 (2016)
29. Christopher, K., Pierre, D.: Stochastic grammatical inference with multinomial tests. In: Grammatical Inference: Algorithms and Applications, pp. 149–160. Springer (2002)
30. Kwiatkowska, M., Norman, G., Parker, D.: The PRISM benchmark suite. In: Proceedings of the 9th International Conference on Quantitative Evaluation of SysTems (QEST' 12), pp. 203–204. IEEE CS Press (2012)
31. Kwiatkowska, M., Norman, G., Parker, D.: PRISM DTMC benchmark models. http://www.prismmodelchecker.org/benchmarks/. Accessed 23 Apr 2018
32. Kwiatkowska, M., Norman, G., Parker, D.: Prism: Probabilistic symbolic model checker. In: Computer Performance Evaluation: Modelling Techniques and Tools, pp. 200–204. Springer (2002)
33. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: International Conference on Runtime Verification, pp. 122–135. Springer (2010)
34. Mao, H., Chen, Y., Jaeger, M., Nielsen, T.D., Larsen, K.G., Nielsen, B.: Learning probabilistic automata for model checking. In: 2011 Eighth International Conference on Quantitative Evaluation of Systems (QEST), pp. 111–120. IEEE (2011)
35. Mao, H., Chen, Y., Jaeger, M., Nielsen, T.D., Larsen, K.G., Nielsen, B.: Learning Markov decision processes for model checking. arXiv preprint arXiv:1212.3873 (2012)
36. Mao, H., Chen, Y., Jaeger, M., Nielsen, T.D., Larsen, K.G.: Learning deterministic probabilistic automata from a model checking perspective. Mach. Learn. **105**(2), 255–299 (2016)
37. Mediouni, B.L., Nouri, A., Bozga, M., Bensalem, S.: Improved learning for stochastic timed models by state-merging algorithms. In: NASA Formal Methods Symposium, pp. 178–193. Springer (2017)
38. Mizera, A., Pang, J., Yuan,Q.: ASSA-PBN: a tool for approximate steady-state analysis of large probabilistic Boolean networks. In: Proceedings of the 13th International Symposium on Automated Technology for Verification and Analysis, LNCS. Springer (2015). http://satoss.uni.lu/software/ASSA-PBN/. Accessed 23 Apr 2018
39. Norman, G., Parker, D., Kwiatkowska, M., Shukla, S.: Evaluating the reliability of nand multiplexing with prism. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **24**(10), 1629–1637 (2005)
40. Norman, G., Shmatikov, V.: Analysis of probabilistic contract signing. J. Comput. Secur. **14**(6), 561–589 (2006)
41. Nouri, A., Raman, B., Bozga, M., Legay, A., Bensalem, S.: Faster statistical model checking by means of abstraction and learning. In: International Conference on Runtime Verification, pp. 340–355. Springer (2014)
42. Reiter, M.K., Rubin, A.D.: Crowds: anonymity for web transactions. ACM Trans. Inf. Syst. Secur. (TISSEC) **1**(1), 66–92 (1998)
43. Rohr, C.: Simulative model checking of steady state and time-unbounded temporal operators. In: Transactions on Petri Nets and Other Models of Concurrency VIII, pp. 142–158. Springer (2013)
44. Ron, D., Singer, Y., Tishby, N.: On the learnability and usage of acyclic probabilistic finite automata. In: Proceedings of the Eighth Annual Conference on Computational Learning Theory, pp. 31–40. ACM (1995)
45. Ron, D., Singer, Y., Tishby, N.: The power of amnesia: learning probabilistic automata with variable memory length. Mach. Learn. **25**(2–3), 117–149 (1996)
46. Sen, K., Viswanathan, M., Agha, G.: Learning continuous time Markov chains from sample executions. In: Proceedings of the First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004, pp. 146–155. IEEE (2004)

47. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: Computer Aided Verification, pp. 202–215. Springer (2004)
48. Shmulevich, I., Dougherty, E.R., Zhang, W.: From boolean to probabilistic boolean networks as models of genetic regulatory networks. Proc. IEEE **90**(11), 1778–1792 (2002)
49. SUTD. Secure water treatment testbed. http://itrust.sutd.edu.sg/research/testbeds/secure-water-treatment-swat/. Accessed 23 Apr 2018
50. Tabakov, D., Vardi, M.Y.: Experimental evaluation of classical automata constructions. In: Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2–6, 2005, Proceedings, pp. 396–411 (2005)
51. Valiant, L.G.: A theory of the learnable. Commun. ACM **27**(11), 1134–1142 (1984)
52. Verwer, S., de Weerdt, M., Witteveen, C.: A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data. In: International Colloquium on Grammatical Inference, pp. 203–216. Springer (2010)
53. Wald, A.: Sequential tests of statistical hypotheses. Ann. Math. Stat. **16**(2), 117–186 (1945)
54. Wang, J.: ziqian. https://bitbucket.org/jingyi_wang/ziqian_develop. Accessed 23 Apr 2018
55. Wang, J.: ziqian evaluation. https://bitbucket.org/jingyi_wang/ziqian_evaluation. Accessed 23 Apr 2018
56. Wang, J., Chen, X., Sun, J., Qin, S.: Improving probability estimation through active probabilistic model learning. In: International Conference on Formal Engineering Methods, pp. 379–395. Springer (2017)
57. Wang, J., Sun, J., Qin, S.: Verifying complex systems probabilistically through learning, abstraction and refinement. arXiv preprint arXiv:1610.06371 (2016)
58. Wikipedia. Golden section search. https://en.wikipedia.org/wiki/Golden-section_search. Accessed 23 Apr 2018
59. Younes, H.L.: Verification and planning for stochastic processes with asynchronous events. Technical report, DTIC Document (2005)
60. Younes, H.L.S., Clarke, E.M., Zuliani, P.: Statistical verification of probabilistic properties with unbounded until. In: Formal Methods: Foundations and Applications, pp. 144–160. Springer (2011)
61. Younes, H.L.S., Kwiatkowska, M., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. Int. J. Softw. Tools Technol. Transf. **8**(3), 216–228 (2006)
62. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Computer Aided Verification, pp. 223–235. Springer (2002)
63. Younes, H.L.S., Simmons, R.G.: Statistical probabilistic model checking with a focus on time-bounded properties. Inf. Comput. **204**(9), 1368–1409 (2006)