Learning Markov models for stationary system behaviors

Chen, Yingke; Mao, Hua; Jaeger, Manfred; Nielsen, Thomas Dyhre; Larsen, Kim Guldstrand; Nielsen, Brian

Published in:
NASA Formal Methods

DOI (link to publication from Publisher):
10.1007/978-3-642-28891-3_22

Publication date:
2012

Document Version
Early version, also known as pre-print

Learning Markov models for stationary system behaviors

Yingke Chen, Hua Mao, Manfred Jaeger,
Thomas D. Nielsen, Kim G. Larsen, and Brian Nielsen

Dept. of Computer Science, Aalborg University, Denmark
{ykchen,huamao,jaeger,tdn,kgl,bnielsen}@cs.aau.dk

Abstract. Establishing an accurate model for formal verification of an existing hardware or software system is often a manual process that is both time consuming and resource demanding. In order to ease the model construction phase, methods have recently been proposed for automatically learning accurate system models from data in the form of observations of the target system. Common for these approaches is that they assume the data to consist of *multiple* independent observation sequences. However, for certain types of systems, in particular many running embedded systems, one would only have access to a single long observation sequence, and in these situations existing automatic learning methods cannot be applied. In this paper, we adapt algorithms for learning variable order Markov chains from a *single* observation sequence of a target system, so that stationary system properties can be verified using the learned model. Experiments demonstrate that system properties (formulated as stationary probabilities of LTL formulas) can be reliably identified using the learned model.

1 Introduction

Model-driven development (MDD) is increasingly used for the development of complex embedded software systems. An important component in this process is model checking [1], where a formal system model is checked against a specification given by a logical expression. Often, the complexity of a real system and its physical components, unpredictable user interactions, or even the use of randomized algorithms make the use of complete, deterministic system models infeasible. In these cases, probabilistic system models and methods for probabilistic verification are needed.

However, constructing accurate models of industrial systems is hard and time consuming, and is seen by industry as a hindrance to adopt otherwise powerful MDD techniques and tools. Especially, the necessary accurate, updated and detailed documentation rarely exist for legacy software or 3rd party components. We therefore seek an experimental approach where an accurate high-level model can be automatically constructed or *learned* from observations of a given black-box embedded system component.

Sen et al. [12] proposed to learn system models for verification purposes, based on the *Alergia* algorithm for learning finite, deterministic, stochastic automata [2]. In [8] we developed a learning approach related to that of [12], and

established strong theoretical and experimental consistency results: if a sufficient amount of data, i.e., observed execution runs of the system to be modeled, is available, then the results of model-checking probabilistic linear-time temporal logic (PLTL) properties on the learned model will be good approximations of the results that would be obtained on the true model. Both [12] and [8] assume that learning is based on data consisting of many independent finite execution runs, each starting in a distinguished, unique initial state of the system. In many situations, it will be difficult or impossible to obtain data of this kind: we may not be able to run the system under laboratory conditions where we are free to restart it any number of times, nor may we be able to reset the system to a well-defined unique initial state.

In this paper, therefore, we investigate learning of system models by passively observing a single, ongoing execution of the system, i.e., from data that consists of a single, long observation sequence, which may start at any point in the operation of the system. This scenario calls for different types of models and learning algorithms than used in previous work. The probabilistic system models we are going to construct are *Probabilistic Suffix Automata (PSAs)* [10]. This is a special type of probabilistic finite automaton, in which states can be identified with finite histories of past observations. Since we are constructing models only for the long-run, stationary behavior of a system, we must also limit the model checking of the learned system to such properties as only refer to this long-run behavior, and not to any initial transitions from a distinguished start state. We therefore define *Stationary Probabilistic Linear Time Temporal Logic* (SPLTL) as the specification language for system properties. Roughly speaking, a SPLTL property $S(\varphi)$ specifies the probability that a system run which we start observing at an arbitrary point in time during the stationary, or steady-state, operation of the system satisfies the LTL property φ.

The main contributions of this paper are: we introduce the problem of learning models for stationary system behavior, and adapt an existing learning algorithm for PSAs [10] to this task. We formally define syntax and semantics of SPLTL properties. We conduct experiments which demonstrate that model-checking SPLTL properties on learned models provides good approximations for the results that would be obtained on the true (but in reality unknown) model.

The paper is structured as follows: in Section 2 we introduce the necessary concepts relating to Markov system models, their stationary distributions, and SPLTL. Section 3 describes our method for learning PSAs and Labeled Markov chains (LMCs). Section 4 contains our experimental results. Section 5 includes conclusion and future work.

2 Preliminaries

2.1 Strings and Suffixes

Let Σ denote a (finite) alphabet, and let Σ^* and Σ^ω denote the set of all finite, respectively infinite strings over Σ. The empty string is denoted by e. For any string $s = \sigma_1 \cdots \sigma_i$, where $\sigma_i \in \Sigma$, we use the following notation:

– The longest suffix of s different from s is denoted by $\text{suffix}(s) = \sigma_2 \dots \sigma_i$.
– $\text{suffix}^*(s) = \{\sigma_k \dots \sigma_i | k = 1 \dots i\} \cup \{e\}$ is the set of all suffixes of s.
– A set of strings S is *suffix free*, if for all $s \in S, \text{suffix}^*(s) \cap S = \{s\}$.

2.2 Markov System Models

A *Labeled Markov chain (LMC)* is a tuple $M = \langle Q, \Sigma, \pi, \tau, L \rangle$, where

– Q is a finite set of states,
– $\pi : Q \to [0, 1]$ is an *initial probability distribution* such that $\sum_{q \in Q} \pi(q) = 1$,
– $\tau : Q \times Q \to [0, 1]$ is the *transition probability function* such that for all $q \in Q$, $\sum_{q' \in Q} \tau(q, q') = 1$.
– $L : Q \to \Sigma$ is a *labeling function*

Labeling functions that assign to states a subset of atomic propositions AP can also be accommodated in our framework by assigning $\Sigma = 2^{AP}$. Since a Markov chain defines a probability distribution over sequences of states, an LMC M with alphabet Σ induces a probability distribution P_M^π over Σ^ω through the labeling of the states.

A subset T of Q in LMC M is called *strongly connected* if for each pair (q_i, q_j) of states in T there exists a path $q_0 q_1 \dots q_n$ such that $q_k \in T$ for $0 \leq k \leq n$, $\tau(q_k, q_{k+1}) > 0$, $q_0 = q_i$, and $q_n = q_j$. If Q is strongly connected, then M is said to be strongly connected. A distribution π_M^s is a *stationary distribution* for M if it satisfies

$$\pi^s(q) = \sum_{q' \in Q} \pi^s(q') \tau(q', q). \tag{1}$$

We abbreviate $P_M^{\pi^s}$ with P_M^s. If an LMC M is strongly connected, then M defines a unique stationary distribution.

In this paper we focus on so-called *probabilistic suffix automata* (PSA). A PSA is an LMC extended with a labeling function $H : Q \to \Sigma^{\leq N}$, which represents the history of the most recent visited states (a string over Σ with length at most N). Given the labeling functions L and H, each state q_i is associated with a string $s_i = H(q_i)L(q_i)$ such that, i) the set of strings labeling the states is suffix free, and ii) for any two states q_1 and q_2, if $\tau(q_1, q_2) > 0$, then $H(q_2)$ is a suffix of s_1. For example, for the PSA in Figure 1(b) the set of strings associated with the states is suffix free. Furthermore, e.g., by considering the states q_{sa} and q_{aa} we have that $H(q_{sa}) = s$, $H(q_{aa}) = a$, $L(q_{sa}) = a$ and $L(q_{aa}) = a$ which represent past and current information respectively. Here $H(q_{aa}) = a$ is a suffix of the string sa associated with q_{sa}. The latter case implies that aa is sufficient for identifying q_{aa} rather than saa. Similarly, q_s has two incoming transitions with different histories, but s is sufficient for identifying q_s. For a given $N \geq 0$, the collection of PSAs are denoted by N-PSA, where each state are labeled by a string of at length most N. In the special case, where all strings in a N-PSA is of length N, then the N-PSA is also called an *N-order labeled Markov chain*. An LMC is called *PSA-equivalent* if there exists a PSA M', such that they define the same distribution over Σ^ω ($P_M^s = P_{M'}^s$).

Example 1. The LMC M and the PSA M' in Figures 1(a) and (b) are specified over the same alphabet $\Sigma = \{s, a, b\}$ and define the same probability distribution over Σ^ω. The LMC M is therefore PSA-equivalent, but it is not a PSA since the set of strings associated with states can not be suffix free.

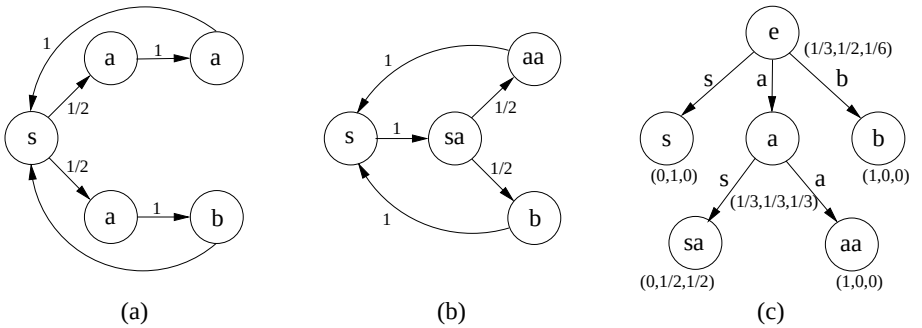

Fig. 1. The LMC in (a) defines the same probability distribution over Σ^ω as the 2-PSA in (b). The probabilistic suffix tree in (c) corresponds to the PSA in (b). The next symbol probabilities associated with the nodes follow the ordering s, a, and b.

The learning algorithm adapted in this paper attempts to find a PSA model that best describes the observed sequence of output symbols generated by a system. However, for the actual learning we will primarily consider an intermediate structure called a *prediction suffix tree* (PST) [10]. A PST over an alphabet Σ is a tree of degree $|\Sigma|$, where each outgoing edge of an internal node is labeled by a symbol in Σ. The nodes of the tree are labeled by pairs (s, γ_s); s is the string defined by labels of the edges on the path from the node in question to the root of the tree. If s' is a descendant of s, then $s \in \text{suffix}^*(s')$. $\gamma_s : \Sigma \to [0, 1]$ is the *next symbol probability function* such that $\sum_{\sigma \in \Sigma} \gamma_s(\sigma) = 1$. The probability that a PST T generates a string $str = \sigma_1\sigma_2 \cdots \sigma_n \in \Sigma^n$ is $\prod_{i=1}^{n} \gamma_{s_{i-1}}(\sigma_i)$, where $s_0 = e$ and s_i is the label of the deepest node reached by following the links corresponding to $\sigma_i\sigma_{i-1} \cdots \sigma_1$ from the root. The PST T in Figure 1(c) shows a representation of the PSA M in Figure 1(b); the node corresponding to the suffix ba is not shown, since the probability of seeing ba is zero. Based on this PST we can, e.g., calculate the probability of seeing the string *sabsaa* from the probabilities of the individual symbols in the string. These probabilities can be found as the next symbol probabilities of the deepest nodes in the tree that can be reached by following (in reverse order) the symbols observed so far. For example, at the root node labeled with the empty string e we have that $\gamma_e(s) = 1/3$. After seeing the string s, the probability of seeing an a is encoded at the node labeled s, where we have $\gamma_s(a) = 1$. The probability of seeing the symbol b following the string sa is encoded at the node labeled sa, where $\gamma_{sa}(b) = 1/2$. Given the string sab, the probability of seeing an s is encoded at the node labeled b, which is the deepest node in the tree reached by following the links corresponding to the symbols bas from the root. By following this procedure for each symbol in the string we get $P(sabsaa) = \gamma_e(s) \cdot \gamma_s(a) \cdot \gamma_{sa}(b) \cdot \gamma_b(s) \cdot \gamma_s(a) \cdot \gamma_{sa}(a) = \frac{1}{3} \cdot 1 \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2}$. See also [10] for further discussion about PSAs and PSTs.

2.3 Stationary Probabilistic LTL

Linear time temporal logic (LTL) [1] over the vocabulary Σ is defined as usual by the syntax

$$\varphi ::= \mathit{true} \mid \sigma \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \mathsf{U} \varphi_2 \qquad (\sigma \in \Sigma)$$

For better readability, we also use the derived temporal operators $\Box$ (always) and $\Diamond$ (eventually).

Let φ be an LTL formula over Σ. For $s = \sigma_0\sigma_1\sigma_2\ldots \in \Sigma^\omega$, $s[j\ldots] = \sigma_j\sigma_{j+1}\sigma_{j+2}\ldots$ is the suffix of s starting with the $(j+1)$st symbol σ_j. The LTL semantics for infinite words over Σ are as follows:

- $s \models \mathit{true}$
- $s \models \sigma$, iff $\sigma = \sigma_0$
- $s \models \varphi_1 \wedge \varphi_1$, iff $s \models \varphi_1$ and $s \models \varphi_2$
- $s \models \neg\varphi$, iff $s \not\models \varphi$
- $s \models \bigcirc \varphi$, iff $s[1\ldots] \models \varphi$
- $s \models \varphi_1 \mathsf{U} \varphi_2$, iff $\exists j \geq 0.\ s[j\ldots] \models \varphi_2$ and $s[i\ldots] \models \varphi_1$, for all $0 \leq i < j$

The syntax of *stationary probabilistic LTL (SPLTL)* now is defined as by the rule:

$$\phi ::= \mathsf{S}_{\bowtie r}(\varphi) \quad (\bowtie \in \geq,\ \leq,\ =;\ r \in [0,1];\ \varphi \in \mathrm{LTL})$$

The syntax of SPLTL, thus, is essentially the same as standard probabilistic LTL (PLTL). However, the semantics will be defined in a slightly different manner. Seen as a PLTL formula, $\mathsf{S}_{\bowtie r}(\varphi)$ would be satisfied by a LMC if traces of the Markov chain satisfy ϕ with probability $\bowtie r$, when initial states of the system are sampled according to the initial state distribution π. In the SPLTL semantics, the unique initial distribution π is replace with the set of all stationary distributions of the Markov chain, and we define for an LMC M:

$$M \models \mathsf{S}_{\bowtie r}(\varphi) \text{ iff } \text{ for all stationary distributions } \pi^s \text{ for } M\text{:} \quad P_M^{\pi^s}(\{s \in \Sigma^\omega | s \models \varphi\}) \bowtie r$$

Note, in particular, that the satisfaction relation $\models$ now only depends on the transition probabilities τ of M, but not on the the initial distribution π.

The reason for this design of SPLTL is that we are interesting in analyzing behaviors of systems that are characterized by an open-ended mode of operation, and which we observe during their ongoing operation. Think, for example, of an elevator control program, a network router, or an online web-service. It will then typically be the case that the system (which may originally have started from some special initial configuration) has reached a terminal strongly connected component of states, and has converged to one of its stationary distributions. Starting to observe the system at a random point in time then corresponds to starting the observation at a state sampled from a stationary distribution. The real-world meaning of $M \models \mathsf{S}_{\bowtie r}(\varphi)$ then is: assuming that we start observing M at a random point in time, but when M is already past a possible initial "burn-in" phase, then the probability that we see the further execution of M having property ϕ is $\bowtie r$.

3 Learning Labeled Markov Chains

Our algorithm for learning PSAs is a modified version of the method described in [10]. The modifications mostly relate to the fact that the data is to be generated by a strongly connected model. As in [10], for a given sample sequence $Seq = \sigma_1\sigma_2 \ldots \sigma_n$, we learn a PSA by firstly constructing a PST T, and then translating T into a PSA. The PSA may be considered as an LMC after removing the labeling function $H(q_i)$ (representing past observations), and the resulting model can then directly be used in probabilistic model checker (PRISM [6] here). The translation from a PST to a PSA is performed as described in [10], and will not be discussed further. The key part in the learning process is the construction of T, which takes the form of a top-down tree-growing procedure. At any point in time, the algorithm maintains a current tree, and a set $\mathcal{S}$ of strings (representing suffixes) that are candidates for inclusion in the tree. In one iteration, the algorithm

(1) selects a string $s \in \mathcal{S}$, and decides whether to add s as a node to T (which may require the addition of intermediate nodes to T that connect s to the leaf in the current T that represents the longest suffix of s contained in T).
(2) (regardless of whether s was added to T) for all $\sigma \in \Sigma$, decide whether to add σs to $\mathcal{S}$.

The crucial question, now, is how exactly to define the decision criteria for (1) and (2). In [10], the decision criteria depend on a parameter as well as a prior specification of both the memory length of the PSA and an upper bound on the number of states of the PSA. The authors prove probably approximately correctness results for the learning algorithm, but the requirement of prior knowledge about model size and memory length is not compatible with our setting. Here we are going to adjust the original criteria by combining parameters and removing prior constraints. As in most statistical learning approaches, a central tool in our learning approach is the *likelihood* of a PST T given the data Seq, i.e., the conditional probability of the data given the model T:

$$L(T \mid Seq) = P(Seq \mid T). \tag{2}$$

We now base both decisions on a single parameter $\epsilon \geq 0$ that is given as input to the PST learning algorithm, and which represents the minimal improvement in likelihood that we want to obtain when adding an extra node to the PST.

For step (1) it is straightforward to compute precisely the improvement in likelihood one will obtain using a tree T containing s as a leaf, compared to the tree T' in which suffix(s) is a leaf (T and T' otherwise being equal), i.e., line 4, in Algorithm 1. We add s to the tree if the improvement is at least ϵ. Exactly the same criterion can not be used in step (2), since here we need to include σs into the candidate set not only when adding σs itself to T leads to a likelihood improvement, but also when this may happen only for some further extension $s'\sigma s$ of σs. However, one can derive a global upper bound on the maximal likelihood improvement obtainable by adding any such $s'\sigma s$,

and we add σs to $\mathcal{S}$ if this bound is at least ϵ, i.e., line 5, in Algorithm 1. The learning algorithm is described in Algorithm 1, where the empirical (conditional) probabilities $\tilde{P}(\cdot)$ are calculated based on the sample sequence Seq.

Algorithm 1 Learn_PSA

Require:
A sample sequence Seq, and the ϵ

Ensure:
A PST $\bar{T}$

1: Initialize $\bar{T}$ and $\mathcal{S}$: let $\bar{T}$ consist of a single root node (corresponding to e), and let $\mathcal{S} = \{\sigma \mid \sigma \in \Sigma \textit{ and } \tilde{P}(\sigma) \geq \epsilon\}$
2: **while** $\mathcal{S} \neq \emptyset$ **do**
3: (A) Pick any $s \in \mathcal{S}$ and remove s from $\mathcal{S}$
4: (B) If

$$\tilde{P}(s) \cdot \sum\nolimits_{\sigma \in \Sigma} \tilde{P}(\sigma|s) \cdot \log \frac{\tilde{P}(\sigma|s)}{\tilde{P}(\sigma|\,\text{suffix}(s))} \geq \epsilon$$

then add s and all its suffixes which are not in $\bar{T}$ to $\bar{T}$
5: (C)If $\tilde{P}(s) \geq \epsilon$, then for every $\sigma' \in \Sigma$, if $\tilde{P}(\sigma' s) \geq 0$, then add $\sigma' s$ to $\mathcal{S}$
6: **end while**
7: Extend $\bar{T}$ by adding all missing sons s of internal nodes if $\tilde{P}(s) > 0$
8: For each $s \in \bar{T}$, let

$$\hat{\gamma}_s(\sigma) = \tilde{P}(\sigma|s')$$

where s' is the longest suffix of s in $\bar{T}$

The learned tree, thus, depends on the value of ϵ. Smaller ϵ lead to the construction of larger trees, and as $\epsilon \to 0$, the size of the tree will typically approach the size of the dataset (because the tree degenerates into a full representation of the data). In Machine Learning terminology, the learned tree then *overfits* the data. In order to avoid overfitting, and to learn an accurate model of the data source, rather than an accurate model of the data itself, one often employs a *penalized likelihood score* to evaluate a model. These scores evaluate candidate models based on likelihood, but subtract a penalty term for the size of the model. Common penalized likelihood scores are *Minimum Description Length* [9] and the *Bayesian Information Criterion (BIC)* [11]. The BIC score of a PSA A relative to data Seq is defined as

$$BIC(A \mid Seq) := log(L(A \mid Seq)) - 1/2 \mid A \mid log(\mid Seq \mid), \tag{3}$$

where $\mid Seq \mid$ is the length of Seq, and $\mid A \mid$ is the number of free parameters which represents the size of model, i.e. $\mid A \mid = \mid Q_A \mid \cdot (\mid \Sigma \mid -1)$. Using a golden section search [14, Section E.1.1] we systematically search for an ϵ value optimizing the BIC score of the learned model.

4 Experiments

In order to test the proposed algorithm we have generated observation sequences from three different system models. We applied the learning algorithm on each

single sampled sequence, and validated resulting models by comparing with the known generating models in terms of their SPLTL properties. For the actual comparison of the models, we considered relevant system properties expressed by LTL formulas as well as a set Φ of randomly generated LTL formulas. Formulas were generated using a stochastic context-free grammar, and each formula was restricted to a maximum length of 30.

In order to avoid generating un-interesting formulas (especially tautologies or unsatisfiable ones), we constructed a dummy model M_d with one state for each symbol in the alphabet, and with uniform transition probabilities. For each generated LTL formula $\varphi \in \Phi$ we tested whether the formula was indistinguishable by the learned model M_l, the generating model M_g, and the dummy model M_d in the sense that $P^s_{M_g}(\varphi) = P^s_{M_l}(\varphi) = P^s_{M_d}(\varphi)$. If that was the case, then φ was removed from Φ.

We compute stationary probabilities of LTL properties using the PRISM model checker [6]; in the experiments performed, all the learned models are strongly connected. PRISM provides algorithms to compute the stationary distribution over the states, and for a given LTL property φ, the probability of φ at any given start state. Combined, this allows us to compute $P^s_M(\varphi)$.

We evaluate the learned models by comparing $P^s_{M_g}(\varphi)$ and $P^s_{M_l}(\varphi)$ for certain properties φ that are of interest for the individual systems, as well as by the mean absolute difference for the random formulas in Φ:

$$D = \frac{1}{|\Phi|} \sum\nolimits_{\varphi \in \Phi} |P^s_{M_g}(\varphi) - P^s_{M_l}(\varphi)| \tag{4}$$

The mean absolute difference between M_g and M_d is calculated analogously. It is denoted D_d, and reported as a reference baseline.

We distinguish experiments in which the data was generated by a PSA-equivalent model, and experiments where the generating model is not exactly representable by a PSA.

4.1 Learning models of PSA-equivalent systems

Phone Model For our first experiment we use a toy model for a telephone. We consider observable state labels i (the phone is idle), r (ringing), t (talking), h (phone is hung up), and p (receiver picked up). The PSA model in Fig. 2 encodes that the probability of a ringing phone being picked up depends on the elapsed time since it has been used last (which can indicate that the phone owner has left in the meantime). To this end, the model has a limited memory for how many time units the phone has been idle since it has last been hung up (or since it has last been ringing without being answered), and, e.g., the probability $P(p|hr)$ is higher than $P(p|hir)$. The model has a memory of histories of at most length 4, but in many cases only a shorter history is relevant for determining the transition probabilities. For example, once the phone is picked up (transition to the state with suffix label p), the previous history becomes irrelevant.

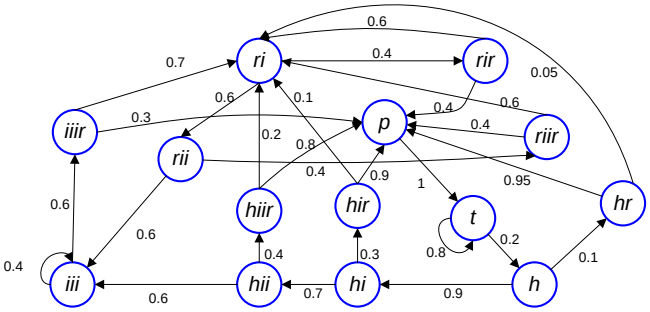

Fig. 2. The phone model. States are labeled by (i)dle,(r)ing, (p)ick-up,(h)ang-up and (t)alk.

Experimental results for this model are summarized in Table 1. The first four columns in the table show: $|Seq|$: the length of the sequence (generated by the model from Figure 2) from which the model was learned; time: time in seconds for the full learning process, including multiple learning runs with different ϵ parameters, and golden section search for optimizing the BIC score; order: the order of the learned model, i.e., the maximal length of a suffix label in the model; $|Q_l|$: number of states in LMC.

The results show that for smaller data sizes a too simple model consisting of a simple Markov chain over the 5 symbols in the alphabet is learned. With more data, the correct structure of the model with its order 4 and 14 states is identified.

Columns 5-10 of Table 1 show the accuracy obtained for checking SPLTL formulas. Column D shows the average error (refer to Equation 4) for 507 random formulas. For comparison: $D_d = 0.1569$. The remaining columns show the stationary probabilities for selected properties of interest. Column t simply contains $P^s_M(t)$, i.e., the long-run frequency of the phone being busy. Column $rp \mid r$ shows the stationary *conditional* probability for the LTL formula $\varphi = r \wedge \bigcirc p$, given that r holds, i.e., the stationary probability that the ringing phone is picked up. Similarly, the next two columns show the probability that the phone is picked up, given that it is ringing, and has been idle for (at least) one, respectively two, time intervals before. Finally, $\Diamond\Box\, i$ is the (unbounded) property that eventually the phone will be idle forever. The results show that the learned models provide very good approximations for the SPLTL properties of the generating model.

Randomized Self-stabilizing Protocol Consider now the randomized self-stabilizing protocol by [4]. This algorithm is designed for ring networks with an odd number of processes, and where each process p_i is equipped with a Boolean variable X_i. The protocol operates synchronously such that if $X_i = X_{i-1}$, then p_i makes a uniform random choice about the next value of X_i; otherwise it sets X_i to the current value of X_{i-1}. Each pair of neighboring processes with the same

Table 1. Experimental results for the phone model

$\|Seq\|$	time(sec)	order	$\|Q_l\|$	D	t	$rp\|r$	$irp\|ir$	$iirp\|iir$	$\Diamond\Box\, i$
80	9.1	1	5	0.0551	0.253	0.333	0.333	0.333	0
160	4	1	5	0.0096	0.370	0.407	0.407	0.407	0
320	6.2	1	5	0.0281	0.344	0.310	0.309	0.309	0
640	6.13	1	5	0.0094	0.392	0.424	0.424	0.424	0
1280	7.5	1	5	0.0064	0.385	0.446	0.446	0.446	0
2560	11.9	1	5	0.0089	0.366	0.447	0.447	0.447	0
5120	36.9	3	10	0.0020	0.379	0.490	0.490	0.490	0
10240	225.2	4	14	0.0014	0.381	0.506	0.477	0.409	0
20480	456.5	4	14	0.0005	0.378	0.515	0.489	0.414	0
M_g		4	14		0.378	0.512	0.488	0.424	0

value assigned to their Boolean variables generates a "token". The network is stable if it only contains a single token. In order to obtain a strongly connected model we have modified the original protocol: after reaching a stable state each process will set its Boolean variable to 0, thus returning to an unstable state.

Using the protocol above we have analyzed the behavior of the learning algorithm by varying the number of processes and the length of the observed sample sequence as well as by changing the level of abstraction. In the first experiment, symbols in the sample sequence correspond to a value assignment to all the Boolean variables associated with processes. Thus, with N processes, there are 2^N symbols in Σ. In the second experiment, we replaced the symbols in the sequence with more abstract labels that only represent the number of tokens defined by the value assignments. For N processes, the alphabet Σ then only contains N symbols.

The results of the experiments are given in Figure 3 and Table 2. Figure 3 shows the probability $P^s_M(true\, \mathsf{U}_{\leq L}\ stable|token = N)$ of reaching a stable configuration within L steps conditional on being in a (starting) configuration where all processes assign the same value to the Boolean variables. In general, we observe a very good fit between the probability values computed for the different models (having the same number of processes). One notable difference is the probability values calculated for the full 7 processes model compared to the abstract and the real 7 processes models. We believe that this discrepancy is due to the length of the sample sequence being insufficient for learning an accurate full model. This hypothesis is supported by the results in Table 2. The table lists the learning time (time), the order of the learned PSA (order), the number of iterations performed by the golden section search (iter), the number of states in LMC ($|Q_l|$), and the average difference in probability (D) calculated according to Equation 4 using 503 random LTL formulas. In particular, we see that with 10240 symbols, the learned full model only contains a single state, whereas the abstract model has four states and a lower average difference in probability. Note, however, that with 50000 symbols the algorithm learns the correct order

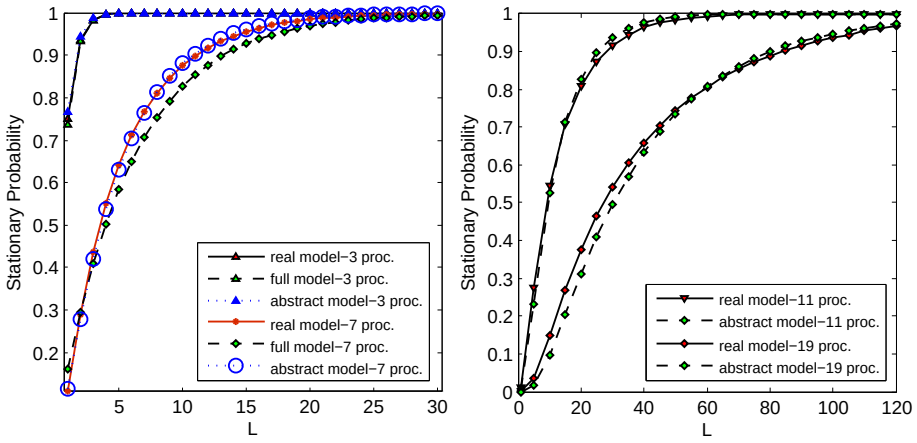

Fig. 3. Left: Experiment results for 3 processes and 7 processes. Right: Experiment results for 11 processes and 19 processes. For 3, 7 and 11 processes, both full and abstract models were learned from 10240 symbols; the abstract 19 process model was learned from 20480 symbols

and number of states for the full model and the average difference in probability becomes significantly smaller.

From Table 2, we also see (as expected) that the time complexity of learning an abstract model is significantly lower than that of learning a full model. Note that due to time complexity, we have not learned full models for networks with 11 and 21 processes.

Since the abstract models are often significantly smaller than the generating models, the time required for model checking using the abstract models is also expected to be lower. We have analyzed this hypothesis further by measuring the time complexity for evaluating the SPLTL property $P^s_M(true\,\mathsf{U}_{\leq L}\ stable|token = N)$ for 19 and 21 processes. For the generating model, the total time is calculated as the time used for compiling the PRISM model description to the internal PRISM representation as well as the time used for the actual model checking. For the abstract model, the total time is calculated as the time used for model learning (which produces a model in the PRISM file format), model compilation, and model checking. Fig. 4 shows the time used by both approaches as a function of L. The time complexity of using the abstract models is close to constant. It consists of a constant time (253 sec. and 284 sec., respectively) for model learning and model compilation, and a negligible additional linear time for model checking.

4.2 Learning models of non PSA-equivalent systems

Consider the LMC in Figure 5(a), which is a modified version of the model by Knuth and Yao [5] that uses a fair coin to simulate the toss of a six-sided die.

Table 2. Experimental results for the self-stabilizing protocol with 7 processes. D is based on 503 random LTL formulas. For reference: $D_d = 0.1669$.

	Full model					Abstract model				
$\|Seq\|$	time(sec)	order	iter	$\|Q_l\|$	D	time(sec)	order	iter	$\|Q_l\|$	D
80	73.0	0	30	1	0.0192	1.6	1	38	4	0.0172
160	49.4	0	23	1	0.0325	2.1	1	41	4	0.0079
320	162.9	0	29	1	0.0292	3.3	1	41	4	0.0369
640	34.3	0	19	1	0.0234	2.3	1	23	4	0.0114
1280	37.2	0	19	1	0.0193	4.1	1	32	4	0.0093
2560	42.0	0	19	1	0.0204	5.0	1	23	4	0.0054
5120	47.9	0	19	1	0.0182	8.9	1	23	4	0.0018
10240	59.3	0	19	1	0.0390	16.3	1	23	4	0.0013
20480	80.7	0	19	1	0.0390	31.4	1	23	4	0.0016
50000	1904.4	1	25	128	0.00034	152.42	1	23	4	0.0011
100k	3435.5	1	25	128	0.00071	308.9	1	23	4	0.0007

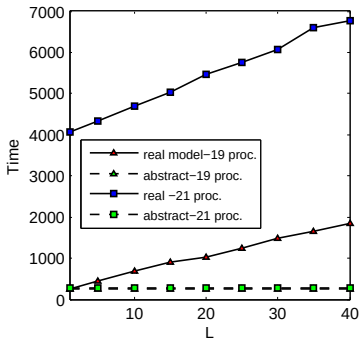

Fig. 4. The time for calculating $P^s_M(true \mathsf{U}_{\leq L}\ stable | token = N)$ (N is the number of process in each model) in the generating model and abstract model. Both abstract models for 19 and 21 processes are learned from a single sequence with 20480 symbols.

For example, *start, H, H, H, T, h2* corresponds to a die toss of 2. Compared to the original model, the model in Figure 5(a) makes a transition back to the start state after having simulated the outcome of a toss.

In this LMC we see that the next symbol probabilities for the two states labeled H on the top branch differ. Specifically, we have that the next symbol probability depends on whether or not we have seen an even or an odd number of Hs, which implies that the model in Figure 5 cannot be represented by any N-order Markov chain and, in particular, any N-PSA. Note that this also implies that the dice model is not PSA-equivalent. An example of a model that was learned from a sample sequence with 1440 observations can be seen in Figure 5(b).

The results of all the experiments are summarized in Table 3. From the table we see that the learned models provide very good approximations for the

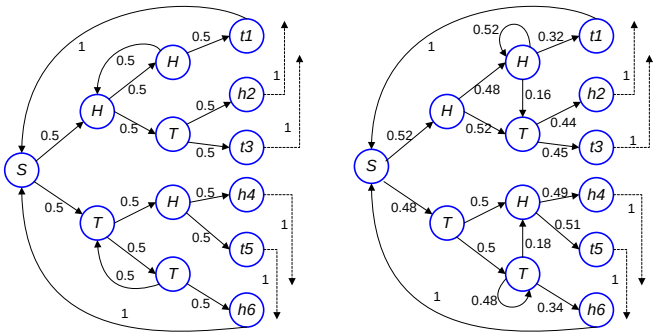

Fig. 5. Dice model. Dash lines will lead to the 'S' state. (a) The generating model. (b) A model learned from a sequence with 1440 symbols.

Table 3. The experiment results for dice model. See table 2 for a description of the columns in the left part of the table. For the right part of the table, D is the mean absolute difference of the learned model and the generating model for stationary probabilities of 501 randomly generated LTL formulas, and $P^s_M(i)$ denotes the stationary probability of getting a i in the next dice toss, and the stationary probability is 1/6 for each number in the generating model.

$\|Seq\|$	time(sec)	order	$\|Q_l\|$	$\|D\|$	$P^s_M(1)$	$P^s_M(2)$	$P^s_M(3)$	$P^s_M(4)$	$P^s_M(5)$	$P^s_M(6)$
360	11.4	2	13	0.0124	0.137	0.17	0.182	0.103	0.205	0.203
720	14.4	2	13	0.0043	0.188	0.174	0.174	0.149	0.168	0.147
1440	16.9	2	13	0.0023	0.184	0.166	0.169	0.143	0.153	0.185
2880	57.4	4	17	0.0023	0.173	0.166	0.159	0.142	0.176	0.184
5760	90.5	4	17	0.0016	0.173	0.165	0.153	0.161	0.174	0.174
11520	159.4	5	19	0.00094	0.162	0.17	0.176	0.157	0.168	0.167
20000	318.4	6	21	0.00092	0.164	0.173	0.171	0.166	0.164	0.162

stationary probabilities of the randomly generated LTL formulas. For example, for the model learned from 20000 observations the mean absolute difference in probability is 0.00092 for 501 random LTL formulas; in comparison, the difference in probability is 0.1014 for the dummy model. A similar behavior is observed for the probability $P^s(i)$ of getting i in the next dice toss.

Finally, we note that the size of the learned model grows as the length of the data sequence increases. This behavior is a consequence of the generating model not being representable by any N-order Markov chain. To illustrate the effect, Figure 6 shows the structure of the model that was learned from 20000 observations. Notice that the differences between the learned model and the generating model relates to the part of the model encoding the number of times we have seen an even number of Hs.

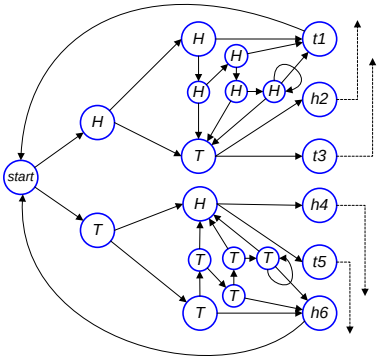

Fig. 6. The dice model learned from a sequence with 20000 symbols. Numbers on edges are omitted.

5 Conclusion

In this paper we proposed to use methods for learning Probabilistic Suffix Automata to learn a formal model for the stationary behavior of a system from a single observation sequence. Compared to previous approaches of learning system models for verification purposes, this extends the scope of applications to scenarios where one can easily obtain data by passively observing a system in its ongoing operation, but where it is difficult to obtain multiple, independent runs under laboratory conditions.

The analysis of the learned model must be restricted to properties that only concern the observed stationary behavior of the system, for which purpose we have introduced SPLTL properties as a suitable specification language. Experimental results show that model-checking SPLTL properties on learned models provides a good approximation to model-checking the true, data-generating system. This can often even be the case when the true system itself is not exactly equivalent to a PSA, in which case the learned model can only be an approximation to the true model.

As in the learning from multiple sequences setting, one could also in the single sequence setting consider methods of *statistical model checking* [15, 13, 7] as an alternative to model learning. While existing statistical model checking approaches are also based on the assumption that data consists of multiple independent system runs, they could be easily adapted to the single sequence case. As discussed in [8], model learning offers several advantages over statistical model checking: learned models also support the model checking of unbounded properties, whereas statistical model checking is limited to bounded properties. Moreover, learned models can support additional analysis and design processes beyond model-checking.

For learning deterministic stochastic automata from multiple sample strings strong consistency results guarantee that in the large sample limit the learned and the true system agree on the probabilities of LTL formulas [2, 8]. Similar

results are not yet established for learning PSAs from a single sequence. Existing results on the consistency of the BIC selection criterion for learning variable order Markov chains [3] strongly indicate that such consistency properties also hold for our learning method, but a full analysis is subject of further work.

References

1. C. Baier and J. P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
2. R. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Computer Science*, pages 139–152. Springer Berlin / Heidelberg, 1994.
3. I. Csiszár and Z. Talata. Context tree estimation for not necessarily finite memory processes, via BIC and MDL. *IEEE Transactions on Information Theory*, 52(3):1007–1016, 2006.
4. T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.
5. D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.
6. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
7. A. Legay and B. Delahaye. Statistical model checking : An overview. *CoRR*, abs/1005.1327, 2010.
8. H. Mao, Y. Chen, M. Jaeger, T. Nielsen, K. Larsen, and B. Nielsen. Learning probabilistic automata for model checking. In *2011 Eighth International Conference on Quantitative Evaluation of Systems*, pages 111 –120, Sept. 2011.
9. J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, Sept. 1978.
10. D. Ron, Y. Singer, and N. Tishby. Power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25:117–149, 1996.
11. G. Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6(2):461–464, 1978.
12. K. Sen, M. Viswanathan, and G. Agha. Learning continuous time markov chains from sample executions. *International Conference on Quantitative Evaluation of Systems,*, 0:146–155, 2004.
13. K. Sen, M. Viswanathan, and G. Agha. Statistical model checkling of black-box probabilistic systems. In R. Alur and D. Peled, editors, *CAV 2004*, 2004.
14. P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2006.
15. H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *CAV 2002,*, 2002.