Statistical Learning of Markov Chains of Programs

Emilio Incerto
IMT School for Advanced Studies
Piazza San Francesco 19
Lucca, Italy
emilio.incerto@imtlucca.it

Annalisa Napolitano
IMT School for Advanced Studies
Piazza San Francesco 19
Lucca, Italy
annalisa.napolitano@imtlucca.it

Mirco Tribastone
IMT School for Advanced Studies
Piazza San Francesco 19
Lucca, Italy
mirco.tribastone@imtlucca.it

***Abstract*—Markov chains are a useful model for the quantitative analysis of extra-functional properties of software systems such as performance, reliability, and energy consumption. However building Markov models of software systems remains a difficult task. Here we present a statistical method that learns a Markov chain directly from a program, by means of execution runs with inputs sampled by given probability distributions. Our technique is based on learning algorithms for so-called variable length Markov chains, which allow us to capture data dependency throughout execution paths by encoding part of the program history into each state of the chain. Our domain-specific adaptation exploits structural information about the program through its control-flow graph. Using a prototype implementation, we show that this approach represents a significant improvement over state-of-the-art general-purpose learning algorithms, providing accurate models in a number of benchmark programs.**

***Index Terms*—statistical learning, quantitative models, extra-functional properties, Markov chains**

I. Introduction

Markov chains (MCs) are a fundamental model to describe the dynamics of computer and communication systems as stochastic processes [5], [45], to enable reasoning about various extra-functional, quantitative properties such as reliability [28], [41], performance [4], and energy consumption [46].

Quantitative properties could also be analyzed through profilers; however, profiling lacks generalizing and predictive power (see also [51]), thus hindering program understanding. On the other hand, Markov modeling hinges on considerable craftsmanship in both the problem domain and in the required analytical techniques, hindering their adoption in practice [50].

A possible solution to mitigate the difficulties in obtaining an accurate MCs from a software system might be to derive it *automatically*. There has been much research into extending higher-level descriptions such as UML diagrams with quantitative annotations (using, for example, appropriate profiles such as MARTE [39]) from which both software artifacts and associated stochastic models are generated [2], [25]. However, this introduces a hard problem of synchronization between the model and the software, since automatically generated code stubs are typically subjected to further manual intervention by the developer [16]. Thus, such model-driven approaches can be particularly difficult to use in general, especially in the context of fast-paced software processes with continuous integration.

Here we aim to bridge this gap by *learning a Markov model from a program*. The starting point is to interpret a program probabilistically by following a well-known argument (see, e.g., [11], [44]) about resolving the nondeterminism of the input by assigning a representative probability law, or modeling intrinsic probabilistic behavior due to the use of primitives that sample values drawn at random from given distributions. However, although such a *probabilistic semantics* of programs is not new [17], [30], it has not yet been used to synthesize probabilistic models for a program as a whole, i.e., without focusing on the quantification of the probability to satisfy specific path formulas [11], [14].

In this paper, we start from the idea that the stochastic behavior of a program can be operationally given as an MC where each state tracks the program location as well as the current values of the program variables (e.g., [44]). Unfortunately, this detailed state representation may make probabilistic analysis infeasible, since the state space grows fast with the size of the domain of the variables.

To cope with this problem, we propose to build a Markov model which tracks program locations while abstracting from variable values. As an example, let us consider the toy snippet on the left, where the branches of the conditional statements are omitted for simplicity since we assume that they do not alter the input variable x; l_0, …, l_6 label the program locations (PLs); x is assumed to be equal to 0 or 1 with equal probability 0.5. Intuitively, one could envisage an MC that only tracks program locations as shown in Fig. 1a, where each conditional statement is represented by a state (here, l_0 and l_3) with two outgoing transitions labelled with the probability of the boolean expression. We refer to it as the *first-order* (i.e., memoryless) MC, since the future behavior only depends on the current state/PL.

```
    int x U{0,1}
l0  if (x == 0)
l1    //if stmt
    else
l2    //else stmt
    end
l3  if (x == 0)
l4    //if stmt
    else
l5    //else stmt
    end
l6
```

It can be easily seen that easily foregoing the information about variable values in the state representation may require knowledge about the history of visited program locations to obtain a precise model. Indeed, the MC in Fig. 1a allows the path $l_0 \rightarrow l_1 \rightarrow l_3 \rightarrow l_5 \rightarrow l_6$ with probability 0.25, but this cannot correspond to any feasible path in the program because the value of variable x tested in line l_3 is not re-sampled (i.e., the truth value of the two branches cannot be discordant).

Higher-order MCs can account for data dependency by next-state probabilities that depend not only on the current PL

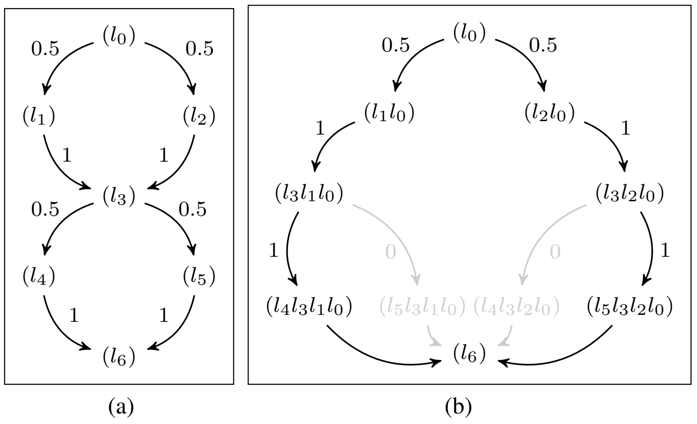


Fig. 1: (a) First-order (i.e., memoryless) and (b) higher-order Markov chain for the example in Section 1.

but also on a subset of the complete history of the program (i.e., the order). A higher-order MC for our example is shown in Fig. 1b. Here each state encodes the history of the program and is labeled with a string of PLs, from the current to the least recent. Such a string represents the *context*, the prefix of a program path that is relevant to determine the probability with which to make transitions. It is now clear to see from Fig. 1b that the model accurately captures the behavior of the original program, i.e., the only admissible paths are $l_0 \to l_1 \to l_3 \to l_4 \to l_6$ and $l_0 \to l_2 \to l_3 \to l_5 \to l_6$, each visited with probability 0.5. In this higher-order MC, interesting properties such as the probability of being in a PL can be recovered from the information from different states; for instance, the probability of being in PL l_3 is equal to the sum of the probabilities for the two states $l_3 l_1 l_0$ and $l_3 l_2 l_0$.

At one extreme, as discussed, is the first-order MC which neglects all previous history. At the other extreme, instead, is a full-order (also called *n-grams* MC) whose state space contains all possible contexts, that is, all possible prefixes of every path of the program. While providing the most precise description, a full-order model exhibits an exponential growth in the number of states, which makes it impractical to use if not for toy examples.

The problem addressed in this paper is to develop an algorithm that learns only the *statistically relevant* contexts for producing an accurate higher-order MC for the program. In particular, we propose the use of *variable length* Markov chains (VLMCs) [8], with a tunable learning algorithm to attain a user-desired trade-off between precision and the amount of history to incorporate in the retained contexts.

Our solution, i.e., *ProgramToVLMC*, is a domain-specific adaptation of a popular algorithm for the statistical learning of VLMCs [31], [33], based on training traces that represent execution paths, and informed by the control-flow graph (CFG) of a program. On a number of benchmarks using a prototype implementation, we show that *ProgramToVLMC* outperforms general-purpose state-of-the-art implementations that would otherwise suffer severe memory or time cost, thus offering a viable solution to derive probabilistic models of programs.

Paper organization: The remainder of this paper is organized as follows. Section II presents related work; Section III provides the foundational concepts about VLMC; Section IV describes the internals of the learning algorithm; in Section V, the numerical evaluation is presented; Section VI discusses relevant open issues and the possible lines of future research.

II. Related Work

a) Code-driven generation of quantitative models: In software performance engineering, the use of MCs is widely accepted (e.g., [13]). However, most of the literature has focused on model-driven approaches [2], while code-driven generation of models has been less explored. Tarvo and Reiss develop a technique for the extraction of queuing network models from a class of multi-threaded programs [47]. They focus on aspects about concurrency and synchronization, but not on detailed models of single tasks, which is instead the objective of the present paper. In [40], an approach for mining the software resource-aware behavior from logs is presented. Although such models could be used to improve the developers' understanding of the system they do not produce an exact analytical representation of the software dynamics. In BEAR [19], MC that model users behaviours are constructed from logs. They do not consider memory effects and they are not concerned with performance. Other works use n-Grams Markov models to improve code autocompletion capabilities or bug finding, such as [22], [48], but they do not focus on performance. Brünink et al. [6] presents an approach that mines a performance model from running systems automatically. However, unlike our approach, it creates performance models by focusing on hot functions (i.e, the most frequently executed ones) only.

b) Code-driven generation of performance profiles: Larus proposes an effective profiling approach for quantifying the impact of the hot portions of program by combining the execution frequencies and measured path cost [27]. Starting from the relative frequency of each path, a probabilistic interpretation of the model could be provided. However, this would be of maximal order since all the paths of the program are explicitly considered. On the contrary, our method is able to produce models that present a high level of accuracy while incorporating only the statistically relevant information.

PerfPlotter analyzes code for performance evaluation [11]. Similarly to our method, this is done by resolving the nondeterminism of a program input probabilistically by assuming a given usage profile through a probability distribution on the input variables. *PerfPlotter* is based on probabilistic symbolic execution [17], and extensions thereof for the reliability analysis of software [14]. It introduces suitable heuristics for the exploration of a program's paths in order to discover exactly the best- and worst-case execution times, together with an approximated version of the whole distribution.

Despite the similar goal, there are significant differences with *PerfPlotter*. First, *PerfPlotter* does not produce a model, but an empirical distribution of a performance metric. From this it is not possible to directly associate performance information to program components. Second, our method explores paths using random sampling, so it may miss best- and worst-case scenarios. However, since the analysis of a program is not tailored to finding these extreme behaviors, the whole distribution can be more accurately captured by our approach in general.

c) Code-driven generation of VLMCs: Approaches that extract VLMCs from code are [35], [36]. There, the focus is on intrusion detection, with a granularity of the model chosen to keep track of sequences of system calls. These VLMCs are built using the general-purpose algorithm in [42], against which we compare in Section V.

d) Probabilistic grammars inference: In [9] the *ALERGIA* algorithm has been proposed for the identification of probabilistic finite automaton (PFA) from sample execution traces. In [42], an equivalence relation and a transformation algorithm between PFA and VLMC have been formally provided showing that a PFA can be seen as a *Markovianitation* of a generating VLMC (i.e., see Section III). However, in [33], it is shown that a PFA could be, in the worst case, quadratically bigger of the respective VLMC, making the usage of VLMC more convenient in practice.

III. Preliminaries

In this section we briefly recall the basic facts about VLMCs [8] instrumental for the development of the paper.

A process shows a variable length memory when the memory length depends on the specific sequence of states that has been observed. The information about which part of the process history is relevant to determine the conditional probability distribution for the next state is encoded in the *context function*, hereby denoted by $\mathcal{C}$. For a sequence of states $x_n x_{n-1} \cdots x_0$, the context $\mathcal{C}(x_n x_{n-1} \cdots x_0)$ is the longest prefix $x_n \cdots x_{n-k+1}$ such that $\Pr(x_i \mid x_n x_{n-1} \cdots x_0) = \Pr(x_i \mid x_n \cdots x_{n-k+1})$ for all states x_i.

To better illustrate this, let us consider a stochastic process with state space $L = \{l_0, l_1\}$, and assume that the conditional probability distribution for the next state depends only on the current state l_j if $j = 0$, but on the previous history if $j = 1$, according to the following context function:

$$\mathcal{C}(x_n \cdots x_0) = \begin{cases} l_0 & \text{if } x_n = l_0, \\ l_1 l_1 & \text{if } x_n = l_1 \wedge x_{n-1} = l_1, \\ l_1 l_0 l_0 & \text{if } x_n = l_1 \wedge x_{n-1} = l_0 \wedge x_{n-2} = l_0, \\ l_1 l_0 l_1 & \text{if } x_n = l_1 \wedge x_{n-1} = l_0 \wedge x_{n-2} = l_1, \end{cases}$$

The set of all contexts can be conveniently represented as a tree, the *context tree*, where each context is encoded as a path ordered from the most to the least recently observed state when traversed from the root to the leaves [42].

Figure 2 shows the context tree of the simple example just described. In this representation, first-level nodes represent 1-length knowledge; thus, a context tree with first-level nodes only is a first order MC. On the contrary, a complete k-height context tree implies a k-order full MC.

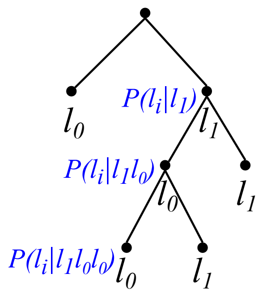


Fig. 2: Context tree enriched with examples of next-symbol probability distributions

A VLMC is obtained by turning the context tree into a *probabilistic suffix tree* (PST) [42], which associates each node with a probability distribution: this represents the *next symbol distribution*, that is, the conditional distribution for the next state given its context. Suppose, for instance, that we wish to find $\Pr(l_i \mid l_1 l_0)$ in the tree of the Figure 2: this is encoded in the probability distribution of the second-level node l_0 that is a direct child of the first-level node l_1.

IV. Learning Algorithm

In this section we show how to learn a VMLC from samples of execution traces of a program by using *ProgramToVLMC*, which is our customized version of the learning algorithm originally proposed in [31], [33].

A. Input Programs

As input we consider any *bounded loop-free* program [21] written in an imperative programming language. That is, we assume that the program has undergone standard transformation techniques such as loop unwinding and function inlining. Specifically, we consider the program's *full* control-flow graph[1] (CFG) $G = (L, E)$, where L is the set of PLs and E is the set of transitions between PLs. Listing 3 reports

```
public void syntheticP(){
l0   Random r=new Random();
l1   int[] b=new int[3];
l2   if(r.nextInt(100)< 20)
l3       b[0]=1;
     else
l4       b[0]=0;
l5   if(r.nextInt(100)< 80)
l6       b[1]=1;
     else
l7       b[1]=0;
l8   if((b[0]==0 && r.nextInt(100)<40)
     ||(b[0]==1 && r.nextInt(100)<90))
l9       b[2]=1;
     else
l10      b[2]=0;
l11  }
```

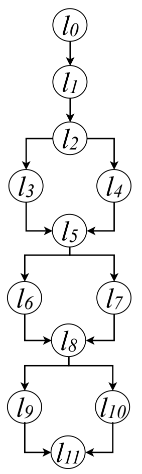


Fig. 3: Simple synthetic program with its control flow

a simple program that will be used as an illustrative example throughout this section, with its CFG.

Capturing the behavior of this program with a stochastic model is not straightforward because of two main difficulties: (i) the behavior is not memoryless, e.g., `b[2]` depends on the evaluation of `b[0]`, which is statements behind the current

[1]That is, the graph whose nodes are all the program's instructions and the edges are control flow transitions among them [18].

Algorithm 1: Fit a VLMC for program P

```
1 ProgramToVLMC (Traces, G, $n_{min}$, $\alpha$)
      inputs : The set of traces; the control-flow of $P$;
               the minimal occurrences of a context; the
               pruning factor
      output: A VLMC for $P$
2     Global sa ← buildSuffixArray(Traces)
3     PST ← $\emptyset$
4     foreach PL $l_i \in G$ do
5         LocationPST ← growPST($l_i$)
6         prune(LocationPST)
7         PST.addChild(LocationPST)
8     end
9     return PST
```

Algorithm 2: Recursively create the location PST of PL s_c by backwardly enlarging the considered context

```
1 growPST ($s_c s_{c-1} \dots s_0$)
      inputs : The input context $s_c s_{c-1} \dots s_0$
      output: The candidate location PST of the PL $s_c$
2     pstNode ← new Node($s_0$)
3     pstNode ← $Pr(s_i | s_c s_{c-1} \dots s_0)$, $\forall s_i | s_c \rightarrow s_i \in E$
4     foreach PL $s_j | s_j \rightarrow s_0 \in E$ do
5         if (isObserved($s_c s_{c-1} \dots s_0 \cup s_j,\ n_{min}$)) then
6             pstNode.addChild(growPST($s_c s_{c-1} \dots s_0 \cup$
                  $s_j$))
7         end
8     end
9     return pstNode
```

one; (ii) the memory has variable length, i.e., `b[0]` and `b[1]` are memoryless conditions while `b[2]` is not.

B. VLMC of a Program

Consider a program with n PLs. The output of the learning algorithm is a VLMC defined through the PST given by the tuple $(\mathcal{C}, \mathcal{P})$ where $\mathcal{C} : \bigcup_{i=1}^{n} L^i \rightarrow \bigcup_{i=1}^{k} L^i$ is the context function and $\mathcal{P} : Im\{\mathcal{C}\} \rightarrow \mathbb{R}^L$ is the next-symbol probability distribution, which associates each context with a conditional probability distribution over the PLs.

C. Learning by ProgramToVLMC

a) Overview: Algorithm 1 describes the construction of a VLMC for a program with CFG G, exercised with a training set of traces. Each trace is a sequence of nodes of G, recorded by executing the program under statistically independent runs. The algorithm is configured by two hyper-parameters that determine the order and accuracy of the learned model: the minimal number of occurrences n_{min} that each context must be found in the traces in order to be used for the generation of the PST, and a pruning factor α. These be detailed later.

First, *ProgramToVLMC* builds the suffix array (cf. [29]) from the traces (line 2); this allows us to efficiently count the number of occurrences of any context, and it will be used in the subsequent stages to efficiently compute all the next-symbol probability distributions. For each location l_i in the CFG, the algorithm builds a candidate sub-PST rooted at l_i which, from now on, we call the *location PST* (line 5). This contains all the prefixes of the CFG paths ending in l_i, since they represent all the possible memory dependencies for that PL. In addition, the function *growPST* computes all the next-symbol probability distributions for the location PST of l_i. Finally, the pruning phase will retain only statistically significant nodes for each location PST (line 6), then add it to the final PST (line 7).

At this stage, building the VLMC by exploiting the structural information of G has one practical advantage: it allows pruning each location PST earlier than general-purpose algorithms, thus reducing the peak memory requirement. See Section V for a numerical evaluation of this aspect.

b) Growing the PST: In the following we will denote a generic context with $s_c s_{c-1} \dots s_0$ where s_c and s_0 are its most and least recent PLs respectively. Algorithm 2 reports the function *growPST*. Its goal is twofold: (i) to create the candidate location PST for s_c, by recursively visiting all of its ancestors, i.e., the PLs from which there exists a path to s_c; and (ii) to compute the corresponding next-symbol probability distributions. To better illustrate this, we detail the base case and the recursive step separately. As an example, we apply Algorithm 2 for generating the location PST of l_5 in Figure 3.

Base step. This represents the first call to the function and is performed by Algorithm 1 at line 5; in our example hence we have $s_c s_{c-1} \dots s_0 \equiv l_5$ as input context. At line 2, the node corresponding to s_0 is created (i.e., since $s_0 = l_5$ this represents the root of the location PST of l_5). Next, in line 3, the next-symbol distribution is calculated considering only l_5 as context, that is, we compute the probabilities $Pr(s_i \mid l_5)$ for all locations. Once calculated, this is stored in the newly created node of the location PST.

Recursive step. In line 4, the recursion starts. The $growPST$ function is invoked on a context enlarged with the PL label of the ancestors; in our example, these are l_3 and l_4. Moreover,at line 5 we limit the exploration to those ancestors whose context has occurred at least n_{min} times in the traces. Once all these recursions are closed, the location PST related to l_5 is returned (line 9) to Algorithm 1 which adds it as a child of the root of the final PST.

c) Learning the next-symbol distribution: The next-symbol probability distribution (see line 3 of Algorithm 2) is computed using the classical frequentist approach: for each context $s_c s_{c-1} \dots s_0$, the probability of visiting the location s_i with that context, $Pr(s_i \mid s_c s_{c-1} \dots s_0)$ is given by the number of sequences containing $s_i s_c s_{c-1} \dots s_0$ divided by the total number of sequences containing $s_c s_{c-1} \dots s_0$. Even this step can benefit from the structural information about the program by limiting the computation of the next-symbol distribution only on the outgoing labels from s_c (of size at most 2), instead of the entire state space L as in the general-purpose learning method of [33].

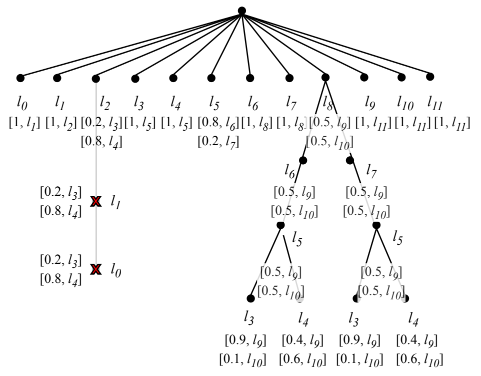


Fig. 4: Learned PST for the Program 3. The red crossed nodes identify pruned nodes in the location PST of l_2.

d) Pruning the tree: The goal of this phase is to eliminate contexts that do not contribute statistically significant memory [8]. For this, the next-symbol probability distribution of each leaf node of the PST is compared with that of its parent and pruned when their *Kullback-Leibler (KL) divergence* [26] is less than a user-specified threshold given by the cutoff $\mathcal{X}^2_{|L|-1;1-\alpha}/2$, where α is the pruning parameter of Algorithm 1. Small α values (i.e., tending to zero) involve smaller PSTs, less memory occupation, and coarser approximations. Instead, high values of α lead to a more faithful reproduction of the program behavior, but also to a greater memory occupation for the PST.

The pruning process is recursively iterated until there are no unexplored leaves (notice that an internal node could become a leaf by possibly pruning all its children). We do not report the pseudo-code for this phase since it has no substantial modification with respect to the state-of-the-art [33].

e) Example: Figure 4 depicts the PST learned for the Program 3 with CFG of Figure 3. Below each node is reported the next-symbol probability distribution of its corresponding context. The values are truncated to the first decimal digit, for readability. Starting from the leaf l_0 the next-symbol probability distribution of the context $l_2 l_1 l_0$ is compared with that of its father context $l_2 l_1$. Since they are equal, the corresponding Kullback-Leibler divergence is zero and l_0 is removed from the tree. The same process is recursively iterated by comparing the probability distribution of the context $l_2 l_1$ with that of the context l_2, which similarly causes the removal of l_1 from this location PST. The red-crossed nodes in Figure 4 identify pruned nodes in the location PST of l_2.

f) Computational Complexity: ProgramToVLMC takes $O(rn^2(\log(rn) + n))$ time, where r and n denote the number of sequences and maximal length of a path of the CGF, respectively (i.e., we have at most n PLs). Although we do not improve the theoretical complexity of the learning process, which is between the state-of-the-art approaches in [8] and [42], thanks to the heuristics defined in this section we are able to extract VLMCs generated by benchmark programs that would not be analyzable through the currently available implementations of the state-of-the-art.

V. Numerical Evaluation

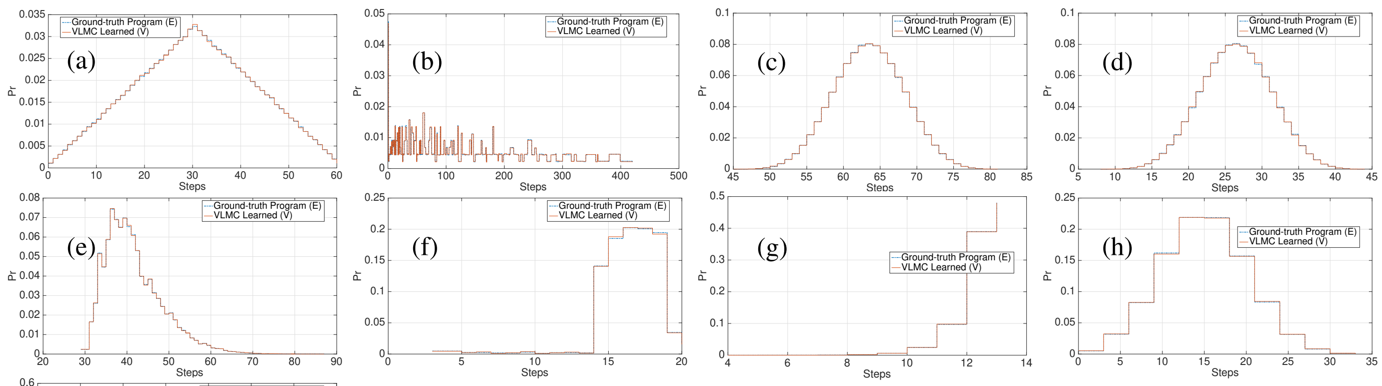
Here we evaluate the accuracy of *ProgramToVLMC* on benchmarks, we conduct a sensitivity analysis with respect to its hyper-parameters, and compare it against the general-purpose VLMC learning techniques from the literature.

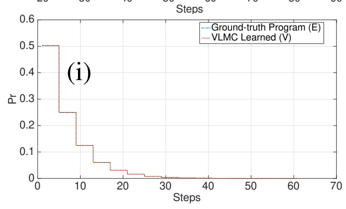
a) Set-up: All experiments were performed on a Linux machine with 48 cores and 60 GB of memory through a prototype Java implementation. The benchmarks are:

- **Double Loop [11]**: It is composed of two loops executed in sequence. For each loop, the number of iterations is random between 0 and 30. We use this as a base case because the program can be formally proven as memoryless. Hence we can evaluate the ability of our approach to produce a VLMC of proper order.
- **Nested Loop [11]**: Two nested loops, each iterating a random number of times between 0 and 20.
- **Bubble Sort, Insertion Sort and Quick Sort [12]**: The input is an array of size 9 with random 32-bit integers.
- **Binary Search [12]**: It implements the binary search that looks for a random generated 32-bit integer y in an array of size 100. The value of y is chosen such that the probability of finding it in the array is 20%.
- **Square root Newton [43]**: It computes the square root of a random double-precision floating point variable between 0 and 100000 using Newton's method.
- **Euclid Algorithm [43]**: It computes the greatest common divisor of two uniformly distributed integer variables between 0 and 200 using Euclid's algorithm.
- **Miller-Rabin [37]**: It implements the Miller-Rabin primality test. Given a uniformly distributed integer variable x between 0 and 1000 it checks if x is prime or not. Differently from the other, it is a randomized algorithm.

For every program, we generated learning traces with uniformly distributed input; however, our approach can be used with any other probability distribution. As quantitative measure of interest, for each benchmark we computed the probability distribution of the number of steps, i.e., the number of locations visited until reaching the final state of the program. To obtain statistically robust estimates, we executed independent samples from each program obtaining independent batches, and stopped when the KL divergence of two successive distributions was less than a threshold (i.e, 10^{-5}). The KL test is meaningful since it is commonly employed to quantify the information loss when one distribution is used in place of another [3]; here, a divergence tending to zero indicates high fidelity of the approximating distribution. We then took the resulting empirical distribution as the "ground-truth" one.

b) Accuracy evaluation: To study the accuracy of Algorithm 1, we ran it with $\alpha = 1$ and $n_{min} = 1$ and compared the ground-truth distribution of the number of steps, E, against the corresponding distribution obtained by simulating the learned VLMC, V, with the same number of runs with which E was generated. We denote the computed KL divergence $D_{KL}(E \parallel V)$ by D_{KL}.





	D_{KL}	Order	#runs
(a) Double Loop	3.63×10^{-5}	31	10^6
(b) Nested Loop	7.58×10^{-4}	22	6×10^5
(c) Bubble Sort	6.45×10^{-5}	84	10^6
(d) Insertion Sort	1.01×10^{-4}	48	5×10^5
(e) Quick Sort	5.35×10^{-5}	81	8×10^5

	D_{KL}	Order	#runs
(f) Binary Search	8.76×10^{-4}	17	2×10^5
(g) Newton's method	1.21×10^{-5}	1	5×10^5
(h) Euclid's GCD	5.95×10^{-5}	1	2×10^5
(i) Miller-Rabin	1.30×10^{-4}	46	10^5

Fig. 5: Comparison between the empirical (ground-truth) and simulated (VLMC-learned) probability distributions of the number of steps until termination, KL divergences, order of the learned VLMC and number of runs for the benchmark programs.

TABLE I: Sensitivity analysis with respect of pruning factor α. For each benchmark we used the same runs of the corresponding ground-truth models (see Figure 5) and $n_{min} = 1$.

	Double Loop			Nested Loop			Bubble Sort			Insertion Sort			Quick Sort		
α	$Order$	$\mathcal{L}$	$\#Nodes$	$Order$	$\mathcal{L}$	$\#Nodes$	$Order$	$\mathcal{L}$	$\#Nodes$	$Order$	$\mathcal{L}$	$\#Nodes$	$Order$	$\mathcal{L}$	$\#Nodes$
1.0	31	77.37	1427	22	407.61	9200	84	717.21	1823996	48	221.91	1116394	81	476.63	1459552
0.9	1	77.37	62	22	408.55	8481	16	726.48	4736	1	221.91	53	40	492.75	20877
0.8	1	77.37	62	22	408.75	8456	16	726.53	4722	1	221.91	53	40	493.26	20515
0.7	1	77.37	62	22	408.88	8438	16	726.62	4704	1	221.91	53	40	493.71	20130

Figure 5 shows that, given a sufficient number of learning samples, *ProgramToVLMC* yields highly accurate models that produce distributions of the number of steps overlapping with the measured ones. The precision is also confirmed by the fact that the KL divergence is always below 10^{-4}. However, it is also possible to observe that *ProgramToVLMC* learns a VLMC of order 31 for the Double Loop program, which was instead supposed to be memoryless by construction. This behavior is due to the choice of $\alpha = 1$, which implies that pruning is allowed only when the compared distributions are equal. This is a too stringent requirement, since $\alpha = 1$ makes the comparison highly sensitive to the noise generated by empirically estimating such distributions by repeating independent runs. Indeed, the confirmation that this choice of α led to an over-sized model is provided by the fact that, by using an infinitesimally less stringent value of α (i.e., $\alpha = 0.999999$), we obtained a memoryless VLMC with $D_{KL} = 7.72 \times 10^{-5}$.

c) Sensitivity evaluation: In this section we study the sensitivity of the proposed technique with respect to its hyper-parameters α and n_{min}.[2] Differently from the previous section, to quantify the accuracy of the learned model here we rely on the notion of log-likelihood function $\mathcal{L}$ of a VLMC, as defined in [32]. We focused on this metric since it is commonly employed for measuring the goodness of fit of a set of candidate statistical models to a sample of data [38]. Lower values of $\mathcal{L}$ lead to high accuracy of the analyzed model, which is therefore preferable among the candidate ones. Moreover, this metric allows the comparison between distributions that do not have the same support, which is instead required for KL divergence. This was needed because the choice of hyper-parameters can have a significant impact on the shape and the support of the learned model.

Table I reports the sensitivity analysis with respect to the pruning factor α, when each model was learned using the same number of runs of the corresponding ground-truth distribution (see Figure 5). Columns *Order* and *#Nodes* give the learned VLMC order and the total number of nodes respectively.

As expected, decreasing values of α tend to considerably smaller size of the learned models, but to larger prediction errors, i.e., higher log-likelihoods. There are, however, cases where smaller α does not impair statistical significance, while causing significant compression of the learned model, as in the case of Double Loop and Insertion Sort. For Double Loop, the results confirm that values of α less than 1.0 yield a memoryless model. Furthermore, under uniformly distributed inputs, Insertion Sort is also memoryless. The other examples

[2]For space reasons, here we report only the most representative benchmarks and choices of hyper- parameters.

TABLE II: Comparison of runtimes (column T) and peak memory usage (column M) of *ProgramToVLMC* and the algorithms [8] and [42] by means of their implementations in [32] and [15], respectively.

	Double Loop					
	ProgramToVLMC		[32]		[15]	
$\#I$	T (s)	M (GB)	T (s)	M (GB)	T (s)	M (GB)
10	5	0.62	46	8.97	168	0.87
15	8	0.70	92	16.00	454	1.15
20	12	0.87	152	21.90	1024	1.43
30	24	1.68	373	40.06	3530	1.90
	Nested Loop					
	ProgramToVLMC		[32]		[15]	
$\#I$	T (s)	M (GB)	T (s)	M (GB)	T (s)	M (GB)
4	5	0.49	1	0.27	229	0.92
5	7	0.60	100	12.98	626	1.16
6	9	0.84	161	19.56	1499	1.49
8	16	0.93	617	45.49	7392	2.19

show variable memory length. As with all statistical methods, *ProgramToVLMC* exhibits a behavior that may depend on the choice of the hyper-parameters. We propose possible strategies for addressing this issue in Section VI.

Comparison against state-of-the-art: Here we evaluate the scalability of our method against general-purpose VLMC learning algorithms originally introduced in [8] and [42], by means of the corresponding implementations available in [32] and [15], respectively.

For this analysis we restricted to two benchmarks, Double Loop and Nested Loop, due to technical limitations of the available implementations. These concern the huge memory consumption; for instance, using [32] requires more than 60 GB of RAM for learning the VLMC of Quick Sort on an array of size 3. Double Loop and Nested Loop were the only two benchmarks usable for a meaningful assessment of scalability with respect to increasing program sizes. Nevertheless, these are representative programs because they have starkly different order of the learned VLMCs. The learning was conducted using the same value for the two hyper-parameters (i.e., $\alpha = 1.00$, $n_{min} = 1$) and with the same number of statistically independent runs (see Figure 5). By manual inspection we verified that all algorithms provided equivalent VLMCs up to small numerical differences due to the use of different statistical tests in the pruning phase.

Table II reports the numerical results where we compare execution time and peak memory occupation of the algorithms when employed for learning VLMCs from the selected benchmarks of increasing complexity, represented by the column $\#I$, which gives the number of iterations with which each loop has been executed. Our approach consistently outperforms the general-purpose approaches for both metrics. We attribute this to the fact that [32] and [15] do not exploit structural information about the program. Indeed, when compared to [32], *ProgramToVLMC* shows considerably less memory consumption. This is because [32] does not limit the maximal order of the learned VLMC; for each context in the collected traces, all the preceding symbols are analyzed in order to discover statistical dependencies. By doing so, a potentially huge number of spurious paths may be explored, such as the ones across two consecutive statistically independent runs. In addition, the pruning phase in [32] is invoked only after the complete creation of the PST. Instead, as discussed in Algorithm 1, *ProgramToVLMC* prunes each location PST, leading to earlier memory release.

We observe that [15] consumes less memory than [32]. This is due to the possibility of specifying a maximal order of the learned VLMC. Still, when compared to our approach, [15] shows considerably longer execution times, although the worst-case computational complexities are similar, as discussed in the previous section. This can be explained by the fact that [15] does not exploit the sparse nature of the VLMC, where each state has at most two transitions, but assumes that the state space is fully connected. As final remark, we stress that albeit [32] works on a fully connected model too, its computation time does not explode with the dimension of the input because of its lower computational complexity.

VI. Conclusion

We have developed a statistical algorithm to derive the VLMC from sample traces of program executions. The model can be used for several types of quantitative analysis, such as estimations of execution times (by computing the probability distribution of the number of steps until termination), hot spots (by finding the program locations with the largest probabilities of being visited), as well as formal queries based, for instance, on probabilistic logics by model checking.

Technically, our main contribution is an adaptation of general-purpose learning algorithms for VLMCs to programs. Despite the promising results, it is possible to identify some limitations which we discuss below.

The required number of samples. We showed that statistical convergence requires a large number of program runs. As a mitigation, it would be possible to parallelize the analysis. Further improvements, which we leave for future work, is to integrate symbolic execution (cf. [24]) with rare event sampling [7], or to use use more sophisticated techniques such as important sampling [20], [23] or concolic testing [34], [49].

Scalability of the learning algorithm. We showed how our method can improve the performance of existing approaches even if it does not improve the theoretical complexity. Although the performance gain is significant both in terms of memory consumption and computing times, extracting VLMCs from complex programs could still be prohibitive since the context are exponential with the number of branches. For taming such an issue, we plan to further improve the computational complexity of the algorithm by following the approach presented in [1] and developing a linear time and space algorithm with respect to the collected traces.

Hyper-parameters tuning. The accuracy of learned VLMC is sensitive to the value of the hyperparameters (i.e., α and n_{min}). Although this issue has been widely discussed in [8], [42], an established methodology has not been proposed. One solution might be to integrate the creation of the VLMC with static program analysis techniques such as model counting [10] or probabilistic symbolic execution [17]. This would allow the precise calculation of path probabilities to guarantee VLMCs of minimal order. We postpone the combination of our approach with the symbolic ones to future work.

REFERENCES

[1] Apostolico, A., Bejerano, G.: Optimal amnesic probabilistic automata or how to learn and classify proteins in linear time and space. Journal of Computational Biology **7**(3-4), 381–393 (2000)

[2] Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Trans. Softw. Eng. **30**(5), 295–310 (2004)

[3] Belov, D.I., Armstrong, R.D.: Distributions of the kullback–leibler divergence with applications. British Journal of Mathematical and Statistical Psychology **64**(2), 291–309 (2011)

[4] Bolch, G., Greiner, S., De Meer, H., Trivedi, K.S.: Queueing networks and Markov chains: modeling and performance evaluation with computer science applications. John Wiley & Sons (2006)

[5] Bolch, G., Greiner, S., de Meer, H., Trivedi, K.: Queueing networks and Markov chains: modeling and performance evaluation with computer science applications. Wiley (2005)

[6] Brünink, M., Rosenblum, D.S.: Mining performance specifications. In: Proc. Int'l Symposium Foundations of Software Engineering (FSE). pp. 39–49 (2016)

[7] Bucklew, J.: Introduction to rare event simulation. Springer Science & Business Media (2013)

[8] Bühlmann, P., Wyner, A.J., et al.: Variable length markov chains. The Annals of Statistics **27**(2), 480–513 (1999)

[9] Carrasco, R.C., Oncina, J.: Learning stochastic regular grammars by means of a state merging method. In: International Colloquium on Grammatical Inference. pp. 139–152. Springer (1994)

[10] Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. Artificial Intelligence **172**(6-7), 772–799 (2008)

[11] Chen, B., Liu, Y., Le, W.: Generating performance distributions via probabilistic symbolic execution. In: Proc. Int'l Conf. Software Engineering (ICSE). pp. 49–60 (2016)

[12] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2009)

[13] Cortellessa, V., Di Marco, A., Inverardi, P.: Model-Based Software Performance Analysis. Springer (2011)

[14] Filieri, A., Păsăreanu, C.S., Visser, W.: Reliability analysis in symbolic Pathfinder. In: Proc. Int'l Conf. Software Engineering (ICSE). pp. 622–631 (2013)

[15] Gabadinho, A., Ritschard, G.: Analyzing state sequences with probabilistic suffix trees: The pst r package. Journal of statistical software **72**(3), 1–39 (2016)

[16] Garcia, J., Krka, I., Mattmann, C., Medvidovic, N.: Obtaining ground-truth software architectures. In: Proc. Int'l Conf. Software Engineering (ICSE). pp. 901–910 (2013)

[17] Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: ISSTA. pp. 166–176 (2012)

[18] Géraud, R., Koscina, M., Lenczner, P., Naccache, D., Saulpic, D.: Generating functionally equivalent programs having non-isomorphic control-flow graphs. In: Nordic Conference on Secure IT Systems. pp. 265–279. Springer (2017)

[19] Ghezzi, C., Pezzè, M., Sama, M., Tamburrelli, G.: Mining behavior models from user-intensive web applications. In: Proceedings of the 36th International Conference on Software Engineering. pp. 277–287 (2014)

[20] Glynn, P.W., Iglehart, D.L.: Importance sampling for stochastic simulations. Management science **35**(11), 1367–1392 (1989)

[21] Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI. vol. 11, pp. 62–73 (2011)

[22] Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software. In: 2012 34th International Conference on Software Engineering (ICSE). pp. 837–847. IEEE (2012)

[23] Jégourel, C., Wang, J., Sun, J.: Importance sampling of interval markov chains. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 303–313. IEEE (2018)

[24] King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976)

[25] Koziolek, H.: Performance evaluation of component-based software systems: A survey. Perf. Eval. **67**(8), 634–658 (2010)

[26] Kullback, S., Leibler, R.A.: On information and sufficiency. The annals of mathematical statistics **22**(1), 79–86 (1951)

[27] Larus, J.R.: Whole program paths. In: ACM SIGPLAN Notices. vol. 34, pp. 259–269. ACM (1999)

[28] Limnios, N., Oprisan, G.: Semi-Markov processes and reliability. Springer Science & Business Media (2012)

[29] Lin, J., Adjeroh, D., Jiang, B.H.: Probabilistic suffix array: efficient modeling and prediction of protein families. Bioinformatics **28**(10), 1314–1323 (2012)

[30] Luckow, K., Păsăreanu, C.S., Dwyer, M.B., Filieri, A., Visser, W.: Exact and approximate probabilistic symbolic execution for nondeterministic programs. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. pp. 575–586 (2014)

[31] Mächler, M., Bühlmann, P.: Variable length markov chains: methodology, computing, and software. Journal of Computational and Graphical Statistics **13**(2), 435–455 (2004)

[32] Maechler, M.: Vlmc: Variable length markov chains. R package version pp. 1–4 (2015)

[33] Magarick, J.: Sequential learning and variable length Markov chains. Ph.D. thesis, Graduate Group in Managerial Science and Applied Economics (2016)

[34] Majumdar, R., Sen, K.: Hybrid concolic testing. In: 29th International Conference on Software Engineering (ICSE'07). pp. 416–426. IEEE (2007)

[35] Mazeroff, G., De, V., Jens, C., Michael, G., Thomason, G.: Probabilistic trees and automata for application behavior modeling. In: 41st ACM Southeast Regional Conference Proceedings (2003)

[36] Mazeroff, G., Gregor, J., Thomason, M., Ford, R.: Probabilistic suffix models for api sequence analysis of windows xp applications. Pattern Recognition **41**(1), 90–101 (2008)

[37] Miller, G., Rabin, M.: Miller-Rabin Primality Test. https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/

[38] Myung, I.J.: Tutorial on maximum likelihood estimation. Journal of mathematical Psychology **47**(1), 90–100 (2003)

[39] Object Management Group: UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE). Beta 1. OMG (2007), OMG document number ptc/07-08-04

[40] Ohmann, T., Herzberg, M., Fiss, S., Halbert, A., Palyart, M., Beschastnikh, I., Brun, Y.: Behavioral resource-aware model inference. In: Proc. Int'l Conf. Automated Software Engineering (ASE). pp. 19–30 (2014)

[41] Pukite, P., Pukite, J.: Markov modeling for reliability analysis. Wiley-IEEE Press (1998)

[42] Ron, D., Singer, Y., Tishby, N.: The power of amnesia: Learning probabilistic automata with variable memory length. Machine learning **25**(2-3), 117–149 (1996)

[43] Sedgewick, R., Wayne, K.: Introduction to programming in Java: an interdisciplinary approach. Addison-Wesley Professional (2017)

[44] Sharir, M., Pnueli, A., Hart, S.: Verification of probabilistic programs. SIAM Journal on Computing **13**(2), 292–314 (1984)

[45] Stewart, W.J.: Performance Modelling and Markov Chains. In: SFM. pp. 1–33 (2007)

[46] Szeliski, R., Zabih, R., Scharstein, D., Veksler, O., Kolmogorov, V., Agarwala, A., Tappen, M., Rother, C.: A comparative study of energy minimization methods for markov random fields with smoothness-based priors. IEEE Transactions on pattern analysis and machine intelligence **30**(6), 1068–1080 (2008)

[47] Tarvo, A., Reiss, S.P.: Automated analysis of multithreaded programs for performance modeling. In: Proc. Int'l Conf. Automated Software Engineering (ASE). pp. 7–18 (2014)

[48] Wang, S., Chollak, D., Movshovitz-Attias, D., Tan, L.: Bugram: bug detection with n-gram language models. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 708–719 (2016)

[49] Wang, X., Sun, J., Chen, Z., Zhang, P., Wang, J., Lin, Y.: Towards optimal concolic testing. In: Proceedings of the 40th International Conference on Software Engineering. pp. 291–302 (2018)

[50] Woodside, M., Franks, G., Petriu, D.C.: The future of software performance engineering. In: Proceedings of the Future of Software Engineering (FOSE). pp. 171–187 (2007)

[51] Zaparanuks, D., Hauswirth, M.: Algorithmic profiling. In: Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI). pp. 67–76 (2012)