$\mathcal{P}$*revent*: a Predictive Run-time Verification Framework Using Statistical Learning

Reza Babaee, Arie Gurfinkel, and Sebastian Fischmeister

Electrical and Computer Engineering, University of Waterloo, Canada

Abstract. Run-time Verification (RV) is an essential component of developing cyber-physical systems, where often the actual model of the system is infeasible to obtain or is not available. In the absence of a model, i.e., black-box systems, RV techniques evaluate a property on the execution path of the system and reach a verdict that the current state of the system satisfies or violates a given property.
In this paper, we introduce $\mathcal{P}$*revent*, a predictive runtime verification framework, in which if a property is not currently satisfied, the monitor generates the probability based on the finite extensions of the execution path, that satisfy the specification property. We use Hidden Markov Model (HMM) to extend the partially observable paths of the system. The HMM is trained on a set of *iid* samples generated by the system. We then use reachability analysis to construct a tabular monitor that provides the probability that the extended path satisfies or violates the specification from the current state. The current state is estimated at run-time using Viterbi algorithm which gives the most probable state. We show the empirical evaluation of $\mathcal{P}$*revent* on a modified version of randomized dining philosopher and on the QNX Neutrino kernel traces collected from the autopilot software of a hexacopter.

1 Introduction

Run-time Verification (RV) [19] has become a crucial element in developing Cyber-Physical Systems (CPSs) [51, 47, 51, 47, 37]. In RV, a monitor checks the current execution, that is a finite prefix of an infinite path, against a given property, typically expressed in Linear Temporal Logic (LTL) [27], that represents a set of acceptable infinite paths. If any infinite extension of a prefix belongs (does not belong) to the set of infinite paths that satisfy the property, the monitor will accept (resp. reject) the prefix. For example, $\varphi_{\mathsf{F}} : \Diamond error$ (resp. $\varphi_{\mathsf{G}} : \Box \neg error$) is satisfied (resp. is not satisfied) on any infinite paths with the prefix $u_1 : \neg error, \neg error, error$. If the monitor is not able to reach a verdict with the prefix, because it can be extended to both satisfying and violating paths, the monitor will output *unknown* [3]. For example, the prefix $u_2 : \neg error, \neg error$ can be extended to both a path that satisfies $\varphi_{\mathsf{F}} : \Diamond error$ (e.g., any extension of u_1) and a path that violates φ_{F} (e.g., $(\neg error)^{\omega}$).

With the exception of *non-monitorable properties* [14] that require an infinite extension, the monitor will be able to reach a verdict, if a finite extension of the prefix is available. A näive approach to extend the finite prefix is to append an infinite sequence of empty string ϵ [4]. In our example, if $\epsilon \models \varphi_{\mathsf{G}}$, by appending ϵ^ω to u_2 we will be able to achieve $u_2 \models \varphi_{\mathsf{G}}$. However, the temporal logic hierarchy [34] of an LTL property implies conflicting semantics for the empty string in combination with X [23]. For instance, $\epsilon \not\models \Diamond error$; otherwise, $\Diamond \mathsf{X} error$ is satisfied on any finite path [23].

In this paper, we propose a different perspective: we estimate the finite extensions of a prefix using a prediction model. The prediction model is trained from identically and independently distributed (*iid*) samples of the previous execution paths of the system, that are collected either on-line or off-line. We use Hidden Markov Models [30] (HMMs) to realize a prediction model of a system with partially observable behavior: the system produces some observations at each state, but the actual state of the system is not directly visible.

In this paper, we merely focus on the properties that can be evaluated with regular extensions, that is, the extensions that are expressible by a Deterministic Finite Automaton (DFA). Depending on the given property, the extensions may specify the prefixes that satisfy the property (*good extensions*) or violate it (*bad extensions*). We use an upper-bound on the length of the estimated extensions. The monitor in our framework, hence, is the result of a bounded reachability analysis on the product of the HMM and the DFA. Using the product model, the monitor is able to predict a verdict, in terms of the probability of the extensions that satisfy or violate the property. To extend an execution path, the monitor needs to know the current state, which is estimated at run-time by Viterbi algorithm [44]. Viterbi algorithm generates the most likely state based on a given observation, i.e., the execution path in our case.

We implemented our approach as a proof-of-concept tool [1], called $\mathcal{P}revent$ (predictive runtime verification framework), and report applying it on two case studies: the original and a modified version of randomized dining philosophers algorithm, and the QNX Neutrino kernel traces collected from running the flight control of a Hexacopter.

Our paper makes the following contributions:

- Introducing $\mathcal{P}revent$, a predictive runtime verification framework to detect satisfaction/violation of a property in advance,
- Constructing a prediction model, that is, the product of a trained HMM and the DFA specifying the good/bad extensions,
- Defining the prediction error on a sequence and evaluating the monitor's performance using hypothesis testing,
- Implementing the runtime monitoring algorithm using Viterbi approximation,
- Evaluating $\mathcal{P}revent$ on two case studies: a modified version of the randomized dining philosophers problem and the flight control of a hexacopter.

[1] Available at `https://bitbucket.org/rbabaeecar/prevent/`

The main sections of our paper are organized as follows: in Section 2 we give an overview of $\mathcal{P}revent$. In Sections 4 and 5 we provide the details of respectively, constructing the the monitor, and the run-time monitoring algorithm. We define a measure to assess the prediction accuracy and validate the performance of the monitor using hypothesis testing in Section 6. Finally, we provide the empirical evaluation of $\mathcal{P}revent$ on two case studies in Section 7.

2 An Overview of $\mathcal{P}revent$

The key idea in $\mathcal{P}revent$ is to finitely extend the execution trace using a prediction model, and check the extended path against the specification property. The prediction model is obtained from *iid* sample traces collected from the past executions of the system. The prediction model enables the monitor to estimate the extensions that satisfy or violate the given property within a finite horizon, represented as the maximum length of the finite extensions. This gives the monitor the ability to detect a property violation before its occurrence with a certain confidence.

Fig. 1 demonstrates an overview of $\mathcal{P}revent$, with its two main components *learning* and *monitoring*, each explained in the following.

Learning: We use the sample traces to train HMM using Baum-Welch algorithm [30]. The training samples are collected independently from the system, representing an independent and identical distribution (*iid*). The trained HMM represents the joint distribution of the paths over Σ^* and S^*, where Σ is the observation space and S is the state space of the system.

Monitoring: The monitor in our framework is the result of a bounded reachability analysis on the product of the HMM and the DFA, that specifies the acceptable or unacceptable extensions by the property. The monitor is implemented as a look-up table, where each entry is a composite state that specifies a DFA state, a hidden state in the HMM, and an observation, and the probability that from the current state the system will satisfy or violate a property in a bounded number of steps. The current hidden states maintain a history of the previous observations (the prefix Y in Fig. 1). The monitor updates its estimation of the current state by running the Viterbi approximation to obtain $(\mathcal{H} \times \mathcal{A})_Y$. The output of the monitor is therefore $Pr(\mathcal{H} \times \mathcal{A}_Y \models \Diamond^{\leq h} Accept)$, where h is the finite horizon, or the maximum lengths of the extensions that are estimated by the monitor. Since $\mathcal{H} \times \mathcal{A}$ has a small size, the probability results of this reachability analysis can be computed off-line for all the states of $\mathcal{H} \times \mathcal{A}$, for $1 \leq h \leq H_{MAX}$, and stored in a table. The value of H_{MAX} represents the maximum length of the extensions that the monitor needs to predict in order to evaluate the property, and can be obtained empirically from the execution samples of the system.

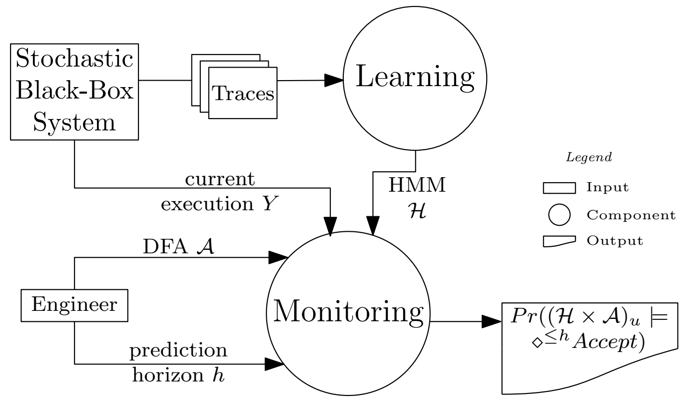


Fig. 1: The overview of $\mathcal{P}revent$ framework.

3 Definitions and Notations

In this section we briefly introduce the definitions and notations that are used throughout the paper.

A probability distribution over a finite set S, is a function $P : S \rightarrow [0,1]$ such that $\sum_{s \in S} P(s) = 1$. We use $X_{1:\tau}$ to denote a sequence $x_1, x_2, \ldots, x_\tau$ of the values of the random variable X, and use u and w to denote generic finite and infinite paths, respectively.

Hidden Markov Model (HMM): HMM is the joint distribution over $X_{1:\tau}$, the sequence of one state variable, and $Y_{1:\tau}$, the sequence of observations (both with identical lengths). The joint distribution is such that $Pr(y_i|X_{1:i}, Y_{1:i}) = Pr(y_i|x_i)$ for $i \in [1..\tau]$ (the current observation is conditioned only on the current state), and $Pr(x_i|X_{1:i-1}, Y_{1:i-1}) = Pr(x_i|x_{i-1})$ for $i \in [1..\tau]$ (the current state is only conditioned on the previous hidden states). We use π to denote the initial probability distribution over the state space, i.e., $Pr(x_1) = \pi(x_1)$. As a result, an HMM can be defined with three probability distributions:

Definition 1 *[HMM] A finite discrete Hidden Markov Model (HMM) is a tuple $\mathcal{H} : (S, \Sigma, \pi, T, O)$, where S is the non-empty finite set of states, Σ is the non-empty finite set of observations, $\pi : S \rightarrow [0,1]$ is the initial probability distribution over the state space, $T : S \times S \rightarrow [0,1]$ is the transition probability between two states, and $O : S \times \Sigma \rightarrow [0,1]$ is the observation probability that the model at each state emits an observation.*

The matrices π, T and O are called the parameters of an HMM, denoted together with Θ.

Discrete-Time Markov Chains (DTMC). We use Discrete-Time Markov Chain (DTMC) to execute the reachability analysis and construct our monitor. A DTMC is defined as follows:

Definition 2 (DTMC) *A (Labelled) Discrete-Time Markov Chain (DTMC) is a tuple* $\mathcal{M} : (S, \Sigma, \pi, \mathbf{P}, L)$*, where* S *is a non-empty finite set of states,* Σ *is a non-empty finite alphabet set,* $\pi : S \rightarrow [0,1]$ *is the initial probability distribution over* S*,* $\mathbf{P} : S \times S \rightarrow [0,1]$ *is the transition probability between two states, such that for any* $s \in S$*,* $P(s,\cdot)$ *is a probability distribution, and* $L : S \rightarrow \Sigma$ *is the labeling function.*

Deterministic Finite Automaton: The extension of a prefix in our setting is described as a Deterministic Finite Automaton (DFA). A DFA is defined as follows:

Definition 3 (DFA) *A Deterministic Finite Automaton (DFA) is a tuple* $\mathcal{A} : (Q, \Sigma,\ \delta, q_I, F)$*, where* Q *is a set of finite states,* Σ *is a finite alphabet set,* $\delta : Q \times \Sigma \rightarrow Q$ *is a transition function determining the next state for a given state and symbol in the alphabet,* $q_I \in Q$ *is the initial state, and* $F \subseteq Q$ *is the set of final states.*

We use $\mathcal{L}(\mathcal{A})$ to show the set of strings accepted by DFA $\mathcal{A}$.

4 Monitor Construction

A monitor is a finite-state machine (FSM) that consumes the output of the system execution sequentially, and produces the evaluation of a given property at each step, typically as a Boolean value [4]. The monitor in our framework is still an FSM, in the form of a look-up table, that instead of Boolean values produces a finite set of values in $[0,1]$. The value indicates the probability of the extensions that satisfy or violate the specification, assuming that the property is currently not satisfied/violated. These probability values are the result of a bounded reachability analysis on the product of the trained HMM and the DFA that specifies the good or bad extensions.

The rest of this section is as follows: in Section 4.1, we describe how an HMM is built using standard *Expectation-Maximization* (EM) learning technique [6], followed by Sections 4.2, which provides details on building a product model as a DTMC that is used to perform the reachability analysis. We finally explain our monitor construction approach in Section 4.3.

4.1 Training HMM

We resort to *Maximum Likelihood Estimation* (MLE) technique to train an HMM. The log-likelihood function $L(\Theta)$ of the HMM $\mathcal{H} : (S, \Sigma, \pi, T, O)$ over an observation sequence $Y_{1:\tau}$ is defined as $L(\Theta) = \log(\sum_{X_{1:\tau}} Pr(X_{1:\tau}, Y_{1:\tau}|\theta))$.

Since the probability distribution over the state sequence $X_{1:\tau}$ is unknown, $L(\Theta)$ does not have a closed form [42], leaving the training techniques to heuristics such as EM. One well-known EM technique for training an HMM is *Baum-Welch* algorithm [30] (BWA), where the training alternates between estimating the distribution over the hidden state variable, $Q : X \to [0, 1]$, with some fixed choice for Θ (*Expectation*), and maximizing the log-likelihood to estimate the values of Θ by fixing Q (*Maximization*) [33].

The *Expectation* phase in BWA computes $Pr(X_t = s|Y, \Theta)$ and $Pr(X_t = s, X_{t+1} = s'|Y, \Theta)$ for $s, s' \in S$ through *forward-backward* algorithm [30]. *Maximization* is performed on a lower bound of $L(\Theta)$ using Jensen's inequality:

$$L(\Theta) \geq Q(X) \log Pr(X_{1:\tau}, Y_{1:\tau}|\Theta) - Q(X) \log(Q(X)) \tag{1}$$

Since the second term is independent of Θ [33], only the first term is maximized in each iteration: $\Theta^{(k)} = \text{argmax}_{\Theta}\, Q(X) \log Pr(X_{1:\tau}, Y_{1:\tau}|\Theta^{(k-1)})$.

The training starts with random initial values for $\Theta^{(0)}$, and consequently running the forward-backward algorithm to update the parameters of the model as follows:

$$\begin{aligned}
\pi^*(s) &= Pr(X_1 = s|Y, \Theta) \\
T^*(s, s') &= \frac{\sum_{t=1}^{\tau} Pr(X_t = s, X_{t+1} = s'|Y, \Theta)}{\sum_{t=1}^{T} Pr(X_t = s|Y, \Theta)} \\
O^*(s, o) &= \frac{\sum_{t=1}^{\tau} \#(Y_t = o) \cdot Pr(X_t = s|Y, \Theta)}{\sum_{t=1}^{T} Pr(X_t = s|Y, \Theta)}
\end{aligned} \tag{2}$$

BWA is essentially a gradient-decent approach, thus its outcome is highly sensitive to the initial values of Θ [42].

We use the Bayesian Information Criterion (BIC) [9] to choose the number of hidden states. BIC assigns a score to a model according to its likelihood but also penalizes models with more parameters to avoid overfitting:

$$BIC(\mathcal{H}) = log(n)|\Theta| - 2L(\Theta) \tag{3}$$

where $|\Theta| = |S|^2 + |S||\Sigma|$ is the size of an HMM, and n is the size of training sample.

4.2 Constructing the Product of the Prediction Model and the Specification

From each state of the trained HMM, the monitor needs to expand the observed execution, u, and determines the evaluation of the given property. The expansion of u is based on a DFA that specifies the good or the bad extensions of u. The monitor maintains the configurations of both the DFA and the trained HMM by creating a product model of both models [46, 50]:

Definition 4 (The Product of an HMM and a DFA) *Let $\mathcal{H} = (S, \Sigma, \pi, T, O)$ and $\mathcal{A} = (Q, \Sigma, \delta, q_I, F)$ respectively be an HMM and a DFA. We define the DTMC $\mathcal{M}_{\mathcal{H}\times\mathcal{A}} : (S' = S \times Q \times \Sigma, \{Accept\}, \pi', \mathbf{P}, L)$ as follows:*

$$\pi'(s,q,o) = \begin{cases} \pi(s) & \textit{if } q \in q_I \\ 0 & \textit{otherwise.} \end{cases}$$

$$\mathbf{P}((s,q,o),(s',q',o')) = \begin{cases} T(s,s') \cdot O(s',o') & \textit{if } \delta(q',o) = q \\ 0 & \textit{otherwise.} \end{cases}$$

$$L(s,q,o) = \begin{cases} \{\textit{Accept}\} & \textit{if } q \in F \\ \emptyset & \textit{otherwise.} \end{cases}$$

4.3 Constructing Monitor with Bounded Prediction Horizon

The monitor's purpose is to estimate the probability of all the finite extensions of length at most h that satisfy a given property. The variable h is a positive integer we call the *prediction horizon*. Let define the finite path $\sigma_0\sigma_1 \ldots \sigma_t$ ($\sigma = (s,q,o) \in S'$ is the composite state of the product model $\mathcal{M}$ and $\sigma_0 = \sigma$, for all $t \leq h$), the extensions according to the given DFA $\mathcal{A}$, such that $L(\sigma_t) = \textit{Accept}$. The monitor's output is $Pr(\sigma_0\sigma_1 \ldots \sigma_t), \forall t \leq h$, which is computed by performing the following reachability analysis on $\mathcal{M}$ [1]:

$$Pr(\sigma \models \Diamond^{\leq h} \textit{Accept}) \tag{4}$$

In order to compute (4), we adopt the transformation of the transition probability in [18]:

$$\mathbf{P}_{Acc}(\sigma,\sigma') = \begin{cases} 0 & \text{if } L(\sigma) = \textit{Accept} \text{ and } \sigma \neq \sigma' \\ 1 & \text{if } L(\sigma) = \textit{Accept} \text{ and } \sigma = \sigma' \\ \mathbf{P}(\sigma,\sigma') & \text{otherwise.} \end{cases} \tag{5}$$

The transformation (5) allows us to recursively compute (4) as follows:

$$Pr(\sigma \models \Diamond^{\leq h} \textit{Accept}) = \sum_{\sigma'} \mathbf{P}_{Acc}(\sigma,\sigma') Pr(\sigma' \models \Diamond^{\leq h-1} \textit{Accept}) \tag{6}$$

Equation (6) is essentially the *transient probability* for $\{\sigma_0 \ldots \sigma_h w\}$ [18], that is, starting from σ_0 the probability of being at state σ_h (i.e., after h steps), such that $L(\sigma_h) = \textit{Accept}$. The probability measure of $\sigma_0 \ldots \sigma_h w$ is based on the prefix $\sigma_0\sigma_1 \ldots \sigma_h$ and can be written as the joint probability distribution of the hidden state variable and that of the observation determined by the underlying trained HMM.

Computing (6) for all the states at runtime is not practical, due to multiplications of large and typically sparse matrices [18]. Instead, for all $t \leq h$ we compute the probabilities off-line and store them in the table $MT(\sigma, t)$, where $MT(\sigma, t) = Pr(\sigma \models \Diamond^{\leq t} Accept)$. Our monitor, thus, is simply transformed into a look-up table with the size at most $O(|S| \times |Q| \times |\Sigma| \times h)$.

5 Run-time Monitoring With Viterbi Approximation

For each state $\sigma = (s, q, o)$ the monitor needs to estimate the hidden state s (q is derivable from o). We employ the Viterbi algorithm to find the most likely hidden state during monitoring.

For an observation sequence $Y = Y_{1:\tau}$ Viterbi algorithm [44, 12] derives $X^*_{1:\tau} = \text{argmax}_{X_{1:\tau}} Pr(X_{1:\tau}|Y, \Theta)$, so-called the *Viterbi path*. Let $v_t(s)$ be the probability of the Viterbi path ending with state s at time t:

$$v_t(s) = O(s, Y_t) \max_{s'}(v_{t-1}(s')T(s', s)) \tag{7}$$

To find X^*_t at step t, the monitor only requires $v_{t-1}(s)$ for all $s \in S$. Therefore, we can obtain X^*_t by using only two vectors (we call *Viterbi vectors*) that maintain the values of $v_t(s)$ and $v_{t-1}(s)$.

Procedure MONITOR demonstrates the runtime monitoring algorithm in $\mathcal{P}revent$. We assume that the monitor table MT is already constructed as described in Section 4 (line **3**). Lines **4-6** initialize the Viterbi vector. The *horizon index* t stores the prediction horizon at each iteration (initialized to h at the beginning–line **8**). Each iteration of the for loop in lines **9-23** is over one observation in the sequence Y. For each observation Y_i, the configuration (s, q, Y_i) (lines **10-11**) combined with t gives us the index to retrieve the probability value in the monitor table (line **12**). If the path is not accepted by the DFA, the monitor will shrink its horizon index by one (t will be decremented—line **16**). Each time that the observed path is accepted by the DFA, the horizon index will be reset to h (line **14**), for the prediction of the next extension. Similarly, once the prediction horizon has reached zero, i.e., the property is not satisfied within the given prediction horizon, the horizon index will be reinitialized to h. At the end, the Viterbi vector is updated for the next iteration in lines **18-22**.

In each monitoring iteration (the loop in lines **9-23**) reading the value from the monitor table MT is constant. For a trained model with k hidden states, updating the Viterbi vector requires $O(k)$ operations of finding maximums (can be improved to $lg(k)$ using a Max-Heap). Therefore, each monitoring iteration is of $O(klg(k))$ in execution time. The space complexity is mainly bounded by the size of the monitor table and the Viterbi vectors: $O(kh)$.

6 Prediction Evaluation

In this section we first define a lower bound on the prediction error of the monitor on a given execution trace, and then use two-sided hypothesis testing to

MONITOR$(Y, \mathcal{H}, \mathcal{A}, h)$

inputs : Execution observation Y, HMM $\mathcal{H} = (S, \Sigma, \pi, T, O)$, DFA $\mathcal{A} = (Q, \Sigma, \delta, q_I, F)$, Prediction Horizon h

output: $Pr((\mathcal{H} \times \mathcal{A})_Y \models \Diamond^{\leq h} Accept)$

begin
    Construct the monitor table $MT(\mathcal{H}, \mathcal{A}, \Sigma, h)$
    foreach $s \in S$ `// Initializing the Viterbi vector`
    do
        $v(s) \leftarrow O(s, Y_1)\pi(s)$
    end
    $i \leftarrow 1$, $t \leftarrow h$, $q \leftarrow q_I$ `//` t `is the horizon index`
    forall $Y_i \in Y$ **do**
        $s \leftarrow \mathrm{argmax}_s\, v(s)$
        $q \leftarrow \delta(q, Y_i)$
        output $MT((s, q, Y_i), t)$ `// Output the prediction`
        if $q \in F$ ***or*** $t = 0$ **then**
            $t \leftarrow h$
        else
            $t \leftarrow t - 1$
        end
        forall $s \in S$ `// Updating the next Viterbi vector`
        do
            $v_{next}(s) \leftarrow O(s, Y_{t+1}) \max_{s'}(v(s')T(s', s))$
        end
        $v \leftarrow v_{next}$, $i \leftarrow i + 1$
    end
end

Runtime monitoring procedure using Viterbi approximation.

evaluate the average prediction performance on a set of testing samples. Finally, we exploit the hypothesis testing results to find an empirical minimum value for the prediction horizon.

6.1 Prediction Error

Let $(o_i \ldots o_{i+\lambda_i(\mathcal{A})})$ be an extension of length $\lambda_i(\mathcal{A})$ at point i that is accepted by a given DFA $\mathcal{A}$, i.e., $(o_i \ldots o_{i+\lambda_i}) \in \mathcal{L}(\mathcal{A})$ (for brevity we use λ_i in the rest of this section). Recall that the monitor's output at point i is the probability of all the extensions of the length at most h that are accepted by $\mathcal{A}$ ($Pr(\sigma_i \models \Diamond^{\leq h} Accept)$). For any $\lambda_i \leq h$ we have:

$$\begin{aligned} Pr(\sigma_i \models \Diamond^{\leq h} Accept) &\geq Pr(\sigma_i \ldots \sigma_{i+\lambda_i} \models Accept) \\ \lambda_i \times Pr(\sigma_i \models \Diamond^{\leq h} Accept) &\geq \lambda_i \times Pr(\sigma \ldots \sigma_{i+k} \models Accept) \end{aligned} \tag{8}$$

We define $\hat{\lambda}_i = \lambda_i \times Pr(\sigma_i \models \Diamond^{\leq h} Accept)$ as the expected value of λ_i estimated by the monitor. Therefore, we can obtain the following minimum error of the prediction at point i:

$$\varepsilon_i^{min} = \lambda_i - \hat{\lambda}_i \tag{9}$$

Notice that since $\lambda_i \geq \hat{\lambda}_i$, ε^{min} is always positive. If there is no k, $i < k < \lambda_i$ such that $(o_i \dots o_{i+k}) \in \mathcal{L}(\mathcal{A})$, i.e., $(o_i \dots o_{i+\lambda_i})$ is the minimal extension that is accepted by $\mathcal{A}$, then $\varepsilon_{i+t}^{min} = (\lambda_i - t) - \hat{\lambda}_{\lambda_i - t}, 0 \leq t < \lambda_i \leq h$, where t is the horizon index in Algorithm Monitor. As a result, the value of ε^{min} can be computed on-the-fly as the monitoring executes.

In our implementation, we assume that there exists at least one point $k \leq h$ such that $(o_i \dots o_{i+k}) \in \mathcal{L}(\mathcal{A})$; otherwise, (9) is not well-defined, and the prediction accuracy can not be verified. If such a point does not exist, we can extend the prediction horizon by increasing h such that there is at least one accepting extension in the trace. The remaining of the path after the last point in which the trace is accepted by $\mathcal{A}$ is discarded as there is no observation to compare the prediction and compute the error.

In the following, we give an empirical evaluation of the monitor's prediction using hypothesis testing which leads to an empirical minimum for h.

6.2 Empirical Evaluation Using Hypothesis Testing

To assess the performance of the prediction, aside from the execution trace, we use hypothesis testing on a set of test samples.

Let $\Lambda = \frac{1}{\tau} \sum_{i=1}^{\tau} \lambda_i$ be the random variable that represents the mean of all λ_i values, for $1 \leq i \leq \tau$. Notice that for *iid* samples, the Λ value of a trace is independent of that of the other traces.

Let $\bar{\lambda}_M$ be the estimation of Λ by the monitor over a set of monitored traces, and $\bar{\lambda}$ be the mean of Λ on a separate set of n *iid* samples with variance ν. We test the accuracy of the prediction using the following two-sided hypothesis test $H_0 : \bar{\lambda}_M = \bar{\lambda}$:

Using confidence α, we use student's t-distribution to test H_0:

$$\frac{\bar{\lambda} - \bar{\lambda}_M}{\frac{\sqrt{\nu}}{n}} \leq t_{n-1,\alpha} \tag{10}$$

Given the mean of the length of the shortest finite extensions in the test sample we can use (10) to obtain a lower bound for h:

$$h \geq \bar{\lambda} - t_{n-1,\alpha} \frac{\sqrt{\nu}}{n} \tag{11}$$

that is, *the prediction horizon* h must be at least as large as the mean of the length of the extensions in the test sample that are accepted by $\mathcal{A}$.

7 Case Studies

We evaluate $\mathcal{P}revent$ on two case studies: (1) the randomized dining philosophers from PRISM case studies [31], which includes the original algorithm, and a modified version that we introduce specifically for evaluating $\mathcal{P}revent$, (2) the QNX Neutrino kernel traces collected from the flight control software of a hexacopter. We show the estimation of *good* and *bad* extensions in the randomized dining philosophers and hexacopter traces, respectively, each of which represents one of the most commonly used property patterns in Matthew Dwyer *et al.* [13]'s survey: *response* pattern in the randomized dining philosophers algorithm, and the *absence* pattern for monitoring a regular safety property [1] in the flight control of a hexacopter. The implementation of monitoring in both experiments is conducted off-line.

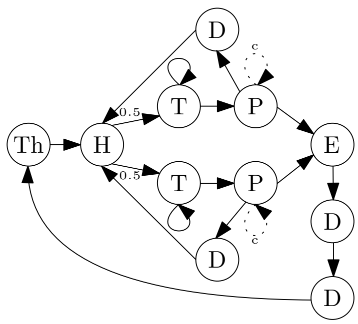
7.1 Randomized Dining Philosopher

We adapt Rabin & Lehmann [29]'s solution to the dining philosophers problem that has the characteristics of a stochastic system to be trained using HMM. We also present a modification of their algorithm, which represents a generic form of decentralized on-line resource allocation [41] that is widely used in distributed and cloud systems [48, 7, 45, 8], wireless communication systems [25, 32, 49], sensor networks [20] and micro-grid management [43]. Our monitoring solution described in Section 7.1 can be particularly considered as a component of the *liveness enforcement supervisory* [22] in such applications.

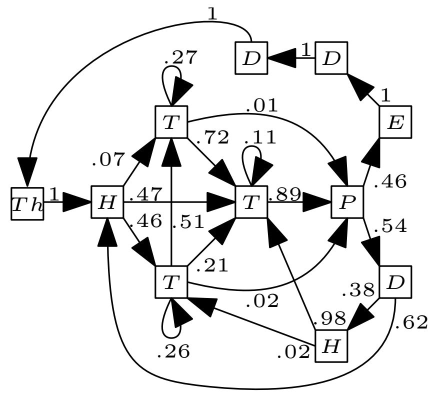
We consider the classic form of the problem, where the philosophers are in a ring topology, and they are selected for execution by a fair scheduler. Fig. 2a demonstrates the state diagram of one philosopher, with Th, H, T, P, D, and E representing the philosopher to be, respectively, *thinking*, *hungry*, *trying*, *picking* a fork, *dropping* a fork, and *eating*. A philosopher starts at (Th), and immediately transitions to (H)[2]. Based on the outcome of a fair coin, the philosopher then chooses to pick the left or the right fork if they are available, and moves to (T). If the fork is not available the philosopher will remain at (T) until it is granted access to the fork. The philosopher will move to (E), if the other fork is available; otherwise, the philosopher will drop the obtained fork, moving to (D), and eventually transitioning back to (H). After the philosopher finishes eating, it will drop (D) the forks in an arbitrary order, and moves back to (Th). This algorithm is shown to be deadlock-free; however, the lockouts are still possible [29].

Our modification of the algorithm is to add a self-transition at (P): a philosopher does not drop the first obtained fork with probability c, i.e., it stays at (P), which is shown with dotted lines in Fig. 2a (the transition from (P) to (D) takes the probability $1-c$, which is not shown in the figure). This modification enables

[2] For simplicity, we remove a self-transition to (Th); however, unlike [11] we do not merge the states (Th) and (H) because we want to distinguish between the incoming transitions to (Th) and (H) in computing the waiting time.



(a) The states of one philosopher in the randomized dining philosophers algorithm. The dotted self-transitions display our modification of the algorithm.



(b) The trained HMM of one philosopher, in a system with three philosophers.

Fig. 2: Training an HMM for the monitored philosopher in a program with three philosophers.

the philosopher to control its *waiting time*, the period between when it becomes hungry for the first time after thinking, and when it eats. A higher value of c means that, instead of going back to (H), the philosopher will more likely stay at (P) so that as soon as the other fork is available it will eat. It is not difficult to observe that as long as there is at least one philosopher with $c \neq 1$, the symmetry that causes the deadlock [29] will eventually break, and the algorithm remains deadlock-free. In a distributed real-time system, where each philosopher represents a process with deadlines, that dynamically change, changing the value of c enables the processes to dynamically adjust their waiting time according to their deadlines.

The purpose of our experiments is to implement a monitor that observes the outputs of a single philosopher, and predicts a potential starvation (lockout) by estimating the extensions that will lead to *eating*.

Predicting Starvation at Run-time We use Matlab HMM toolbox to train HMMs, and 100 *iid* samples collected from the implementation of our modified version, with $c = 1$ for all philosophers except the one that is being monitored[3]. The trained model presents the behavioral signature of the system when a *longer waiting time* is likely. The size of HMM (i.e., the number of hidden states) is chosen based on the *BIC score* of each model with different sizes (see Section 4.1).

[3] We tweaked the implementation in `https://ti.tuwien.ac.at/tacas2015/` from [16].

Fig. 2b demonstrates the trained HMM of one philosopher that is constructed from the traces of a 5-second execution of three philosophers. The trained model reflects the distribution of the prefixes in the training sample, which in turn is determined by how the scheduler behaved during training (i.e., resolving non-determinism of the model) as well as other philosopher. For instance, multiple consecutive *try*s in the training sample will create several states in the trained HMM, each emitting the symbol (T), but only one has a high probability to transition to (P) and the others will model the state where the philosopher can not pick a fork.

The finite extensions that we consider in the prediction are based on the following regular expression: $(\neg hungry)^*(hungry(\neg eat)^* eat(\neg hungry)^*)^*$.

Fig. 3 gives a comparison between the prediction results ($h = 33$) of two trained models, one trained using the samples from the original implementation (LR) and the other one trained from the samples of our modified version (LR-sap), both containing three philosophers. The monitored trace is synthesized in a way that it does not contain any *eat*, and up to point 33 the philosopher is only at state (T). After that the philosopher frequently picks and drops a fork. When the last event of a prefix is *pick*, compared to when it ends with any other observations, the philosopher will have a higher chance to reach *eat* (e.g., with probability 0.98 at point 35); however, since HMM maintains the history of the trace, a prefix with frequent *pick drop* one after another shows a decline in the probability of observing *eat* (e.g., with probability of 0.8 at point 57). These results are more informative than the näive way of extending the path. The prediction results in Fig. 3 also demonstrate that the model that is trained on the *bad* extensions provides an under-approximation for the model that is trained on the *good* traces, and is more conservative in predicting a potential starvation, and thus, produces more false positives.

The summary of our results is displayed in Table 1. We use PRISM to perform the reachability analysis on the product of the trained HMM and DFA. The size of the product model is equal to the size of the HMM, as each state in the trained HMMs emits exactly one observation. The *minimum prediction horizon* (h^{min}) is obtained empirically from 100 test samples. We choose the prediction horizon to be three times as large as h^{min} during monitoring. The average of the estimated length of the acceptable extensions by the monitor is shown as $\bar{\lambda}_M$, and the mean of the error on the entire testing set is denoted by $mean(\varepsilon^{min})$. In average, the monitor predicts the next *eat* (within the prediction horizon) with one step error. The monitor is not able to detect the waiting periods that approximately are longer than $3 \times h^{min} \pm 1$. Increasing the prediction horizon will decrease the error, with the cost of a larger monitor table (MT). The value of $\bar{\lambda}_M$ demonstrates the number of the discrete events produced by the monitored philosopher. With more philosophers $\bar{\lambda}_M$ decreases because the monitored philosopher, and hence the monitor, are scheduled less often when there are more philosophers.

Table 1: Prediction results on 100 test samples.

N	Size of HMM	BIC (+e03)	h^{min}	Size of MT	$\bar{\lambda}_M$	$mean(\varepsilon^{min})$
3	17	25.1	9.94	360	9.30	1.75
4	14	11.9	5.49	180	5.30	1.28
5	10	10.1	6.36	154	6.16	0.80
6	14	7.69	5.61	180	5.17	1.05
7	16	6.09	4.28	170	3.84	1.06
8	10	5.42	4.94	110	4.32	1.33
9	14	4.83	3.15	120	2.77	0.92
10	10	4.40	4.31	110	3.84	0.97

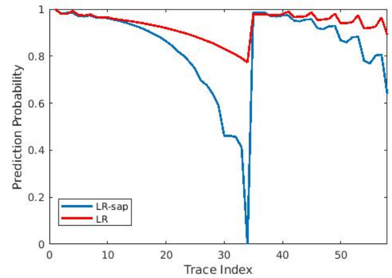


Fig. 3: The comparison of the prediction results from two trained models.

7.2 Hexacopter Flight Control[4]

In this section, we apply $\mathcal{P}revent$ to detect injected faults from QNX Neutrino's [28] kernel calls. The traces are obtained using QNX `tracelogger` during the flight of a hexacopter. The vehicle is equipped with an autopilot, but can be controlled manually using a remote transmitter. The autopilot system uses a cascaded PID controller. QNX's microkernel follows message-passing architecture, where almost all the processes (even the kernel processes) communicate via sending and receiving messages that are handled by the kernel calls `MSG-SENDV`, `MSG-RECEIVEV`, and `MSG-REPLY`. Fig. 4a shows a sub-trace of our the kernel calls sample from the hexacopter.

In this case study, we inject faults by introducing an interference process, with the same priority as the autopilot process, that simply runs a while-loop to consume computation time. The interference process abrupts message-passing between the processes of the same or lower priorities, causing a kernel call to handle the error (typically due to a timeout) and to unblock the sender (shown as event `MSG_ERROR` in Fig. 4a). The purpose of the monitor is to predict the existence of an interference process by only observing the kernel calls.

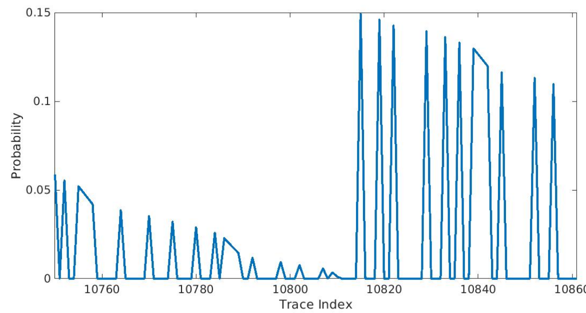
We use SFIHMM [10] on an Intel Xeon 2.40GHz 128GB RAM machine with Debian 9.3 to train an HMM from 1-second of the auto-pilot execution, with the intervening process in full effect. The HMM with the minimum BIC has 19 states. The regular expression $(\neg\texttt{MSG\_ERROR})^*$ $(\texttt{MSG\_ERROR})\Sigma^*$ is used to generate the finite extensions that contain an instance of `MSG_ERROR` (i.e., the bad prefixes of the property $\Box\neg\texttt{MSG\_ERROR}$).

The monitor's prediction on part of the trace generated from another scenario where the interference process started executing in the middle of the flight, is depicted in Fig. 4. The event `MSG_ERROR` is emitted at index 10861, and the probability of the prefix that contains `MSG_ERROR` within next 50 steps is 0.15 at index 10815. The points where the probability is *zero* is because the monitor

[4] Full system description is available at `https://wiki.uwaterloo.ca/display/ESGDAT/QNX+Hexacopter+Flight+Control+Dataset`

⋮
10843 : `MSG-SENDV`
10844 : `MSG-SEND-PULSE`
10845 : `MSG-REPLYV`
10846 : `MSG-RECEIVEV`
10847 : `MSG-RECEIVEV`
10848 : `MSG-RECEIVEV`
10849 : `MSG-RECEIVEV`
10850 : `MSG-SENDV`
10851 : `CONNECT-CLIENT-INFO`
10852 : `MSG-REPLYV`
10853 : `MSG-RECEIVEV`
10854 : `MSG-SENDV`
10855 : `CONNECT-CLIENT-INFO`
10856 : `MSG-REPLYV`
10857 : `MSG-RECEIVEV`
10858 : `MSG-RECEIVEV`
10859 : `MSG-SENDV`
10860 : `CONNECT-CLIENT-INFO`
10861 : `MSG-ERROR`
⋮

(a) A sub-trace of the kernel calls, 20 steps before the event `MSG_ERROR`.



(b) The monitor prediction 50 steps before the event `MSG_ERROR`.

Fig. 4: The monitoring of $\Box\neg$`MSG_ERROR` on the flight control trace with the interference process.

was not able to correctly estimate the hidden state of the model. More training samples are required to enable the monitor to estimate the correct state of the model (in our case for example, three consecutive instances of `MSG_RECEIVEV` have not appeared in the training sample, hence, the prefix can not be associated to any state of the model by the monitor).

8 Related Work

There have been numerous proposals to define semantics of LTL properties on the finite paths [23]; however, to the best of our knowledge, this paper is the first approach in verifying finite paths based on the extensions obtained from a trained HMM.

HMMs have been recently used in run-time monitoring of CPSs [40, 15, 36, 38, 47, 35, 2]. Prasad Sistla *et al.* [36] propose an *internal* monitoring approach (i.e., the property is specified over the hidden states) using specification automata and HMMs with infinite states. Learning an infinite HMM is a harder problem than the finite HMMs, but does not require inferring the size of the model [5].

The notion of *acceptance accuracy* and *rejection accuracy* in [35] are the complement to our notion of prediction error. According to their definition, our Viterbi approximation generates a threshold *conservative* monitor for any regular safety property and regular finite horizon. The analytical method to find an upper bound for the timeliness of a monitor [38] can be applied to $\mathcal{P}revent$ to find an upper bound for h.

Several works focus on efficiently estimating the internal states of an HMM at runtime using particle filtering [40, 15]. Particle filtering uses weights based on the number of particles in each state, and updates the weights in each observation. Viterbi algorithm provides the most likely state, as an over-approximation. Adaptive Runtime Verification [2] couples state estimation [40] with feed-back control loop to generate several monitors that run on different frequencies. These works are orthogonal to our framework and can be combined with $\mathcal{P}revent$.

Learning models for verification is executed on Markov Chain models [26, 21]. HMMs are trained in [16] for statistical model checking. Our work focuses on predictive monitors using a similar technique. We also provide assessments for evaluating the learned model and inferring its size.

9 Conclusion

We introduced $\mathcal{P}revent$, a predictive run-time monitoring framework for properties with finite regular extensions. The core part of $\mathcal{P}revent$ involves learning a model from the traces, and constructing a tabular monitor using reachability analysis. The monitor produces a quantitative output that represents the probability that from the current state, the system satisfies a property within a finite horizon. The current state is estimated using Viterbi algorithm. We defined an empirical evaluation of the prediction using the expected length of the extension of the execution that satisfies the property. In future, we are interested in exploring other evaluation methods, including comparing the prediction results of the trained model with those of the *complete* model by applying *abstraction* [17].

We provided preliminary evaluation of our approach on two case studies: the randomised dining philosophers problem, and the flight control of a hexacopter. In both cases, the trained models are extracted from *bad* traces, thus, the monitor has a tendency to produce false positives. An interesting future modification to our approach, which reduces the number of false positives, is to involve a mixture of trained models based on good and bad traces. Only those models are employed in prediction, that have high correlation with the past observation (i.e., a higher likelihood of the generated prefix).

Lastly, an implementation of $\mathcal{P}revent$ with the application of on-line learning methods (such as state merging or splitting techniques [39, 24]) is necessary to apply the framework to the real-world scenarios.

References

1. C. Baier and J. Katoen. *Principles of model checking.* MIT Press, 2008.
2. E. Bartocci, R. Grosu, A. Karmarkar, S. A. Smolka, S. D. Stoller, E. Zadok, and J. Seyster. Adaptive runtime verification. In *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, pages 168–182, 2012.

3. A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*, pages 126–138, 2007.
4. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *J. Log. Comput.*, 20(3):651–674, 2010.
5. M. J. Beal, Z. Ghahramani, and C. E. Rasmussen. The infinite hidden markov model. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, pages 577–584, Cambridge, MA, USA, 2001. MIT Press.
6. J. A. Bilmes. A gentle tutorial of the EM algorithm and its applications to parameter estimation for Gaussian mixture and hidden Markov models. Technical Report TR-97-021, International Computer Science Institute, Berkeley, CA, 1997.
7. A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. In K. Jeffay, I. Stoica, and K. Wehrle, editors, *Quality of Service — IWQoS 2003*, pages 381–398, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
8. M. Choy and A. K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 593–602, New York, NY, USA, 1992. ACM.
9. G. Claeskens and N. L. Hjort. *Model Selection and Model Averaging.* Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2008.
10. S. DeDeo. Conflict and computation on Wikipedia: A finite-state machine analysis of editor interactions. *Future Internet*, 8(3):31, 2016.
11. M. Duflot, L. Fribourg, and C. Picaronny. Randomized dining philosophers without fairness assumption. *Distributed Computing*, 17(1):65–76, 2004.
12. R. Durbin, S. R. Eddy, A. Krogh, and G. J. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, 1998.
13. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 411–420, 1999.
14. Y. Falcone, J. Fernandez, and L. Mounier. Runtime verification of safety-progress properties. In *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*, pages 40–59, 2009.
15. K. Kalajdzic, E. Bartocci, S. A. Smolka, S. D. Stoller, and R. Grosu. Runtime verification with particle filtering. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, pages 149–166, 2013.
16. K. Kalajdzic, C. Jégourel, A. Lukina, E. Bartocci, A. Legay, S. A. Smolka, and R. Grosu. Feedback control for statistical model checking of cyber-physical systems. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, pages 46–61, 2016.
17. J. Katoen. Abstraction of probabilistic systems. In *Formal Modeling and Analysis of Timed Systems, 5th International Conference, FORMATS 2007, Salzburg, Austria, October 3-5, 2007, Proceedings*, pages 1–3, 2007.

18. M. Z. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In *Formal Methods for Performance Evaluation, 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures*, pages 220–270, 2007.
19. M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
20. G. Mainland, D. C. Parkes, and M. Welsh. Decentralized, adaptive resource allocation for sensor networks. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 315–328, Berkeley, CA, USA, 2005. USENIX Association.
21. H. Mao, Y. Chen, M. Jaeger, T. D. Nielsen, K. G. 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.
22. J. Moody and P. Antsaklis. *Supervisory Control of Discrete Event Systems Using Petri Nets*. The International Series on Discrete Event Dynamic Systems. Springer US, 2012.
23. A. Morgenstern, M. Gesell, and K. Schneider. An asymptotically correct finite path semantics for LTL. In *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, pages 304–319, 2012.
24. K. Mukherjee and A. Ray. State splitting and merging in probabilistic finite state automata for signal representation and analysis. *Signal Processing*, 104:105–119, 2014.
25. D. Niyato and E. Hossain. Competitive pricing for spectrum sharing in cognitive radio networks: Dynamic game, inefficiency of nash equilibrium, and collusion. *IEEE Journal on Selected Areas in Communications*, 26(1):192–202, Jan 2008.
26. A. Nouri, B. Raman, M. Bozga, A. Legay, and S. Bensalem. Faster Statistical Model Checking by Means of Abstraction and Learning. In *RV*, Toronto, Canada, Sept. 2014.
27. A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977.
28. Qnx neutrino rtos. `http://blackberry.qnx.com/en/products/neutrino-rtos/neutrino-rtos`. Accessed: 2017-08-14.
29. M. O. Rabin and D. Lehmann. The advantages of free choice: A symmetric and fully distributed solution for the dining philosophers problem. In A. W. Roscoe, editor, *A Classical Mind*, pages 333–352. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.
30. L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.
31. Radomised dining philosophers case study. `http://www.prismmodelchecker.org/casestudies/phil.php`. Accessed: 2018-01-24.
32. P. Ramanathan, K. M. Sivalingam, P. Agrawal, and S. Kishore. Dynamic resource allocation schemes during handoff for mobile multimedia wireless networks. *IEEE Journal on Selected Areas in Communications*, 17(7):1270–1283, Jul 1999.
33. S. T. Roweis and Z. Ghahramani. A unifying review of linear gaussian models. *Neural Computation*, 11(2):305–345, 1999.
34. K. Schneider. Improving automata generation for linear temporal logic by considering the automaton hierarchy. In *Logic for Programming, Artificial Intelligence,*

and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings, pages 39–54, 2001.
35. A. P. Sistla and A. R. Srinivas. Monitoring temporal properties of stochastic systems. In *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings*, pages 294–308, 2008.
36. A. P. Sistla, M. Zefran, and Y. Feng. Monitorability of stochastic dynamical systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 720–736, 2011.
37. A. P. Sistla, M. Zefran, and Y. Feng. Runtime monitoring of stochastic cyber-physical systems with hybrid state. In *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, pages 276–293, 2011.
38. A. P. Sistla, M. Zefran, Y. Feng, and Y. Ben. Timely monitoring of partially observable stochastic systems. In *17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC'14, Berlin, Germany, April 15-17, 2014*, pages 61–70, 2014.
39. A. Stolcke and S. M. Omohundro. Best-first model merging for hidden markov model induction. *CoRR*, abs/cmp-lg/9405017, 1994.
40. S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok. Runtime verification with state estimation. In *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, pages 193–207, 2011.
41. A. S. Tanenbaum and M. van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007.
42. S. Terwijn. On the learnability of hidden markov models. In *Grammatical Inference: Algorithms and Applications, 6th International Colloquium: ICGI 2002, Amsterdam, The Netherlands, September 23-25, 2002, Proceedings*, pages 261–268, 2002.
43. J. S. Vardakas, N. Zorba, and C. V. Verikoukis. A survey on demand response programs in smart grids: Pricing methods and optimization algorithms. *IEEE Communications Surveys Tutorials*, 17(1):152–178, Firstquarter 2015.
44. A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Information Theory*, 13(2):260–269, 1967.
45. W. Wei, X. Fan, H. Song, X. Fan, and J. Yang. Imperfect information dynamic stackelberg game based resource allocation using hidden markov for cloud computing. *IEEE Transactions on Services Computing*, PP(99):1–1, 2017.
46. C. M. Wilcox and B. C. Williams. Runtime verification of stochastic, faulty systems. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, pages 452–459, 2010.
47. A. Yavolovsky, M. Zefran, and A. P. Sistla. Decision-theoretic monitoring of cyber-physical systems. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, pages 404–419, 2016.
48. Y. O. Yazir, C. Matthews, R. Farahbod, S. Neville, A. Guitouni, S. Ganti, and Y. Coady. Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 91–98, July 2010.
49. C. D. Young. Usap multiple access: dynamic resource allocation for mobile multihop multichannel wireless networking. In *MILCOM 1999. IEEE Military Communications. Conference Proceedings (Cat. No.99CH36341)*, volume 1, pages 271–275 vol.1, 1999.

50. L. Zhang, H. Hermanns, and D. N. Jansen. Logic and model checking for hidden markov models. In *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, pages 98–112, 2005.
51. X. Zheng, C. Julien, R. Podorozhny, F. Cassez, and T. Rakotoarivelo. Efficient and scalable runtime monitoring for cyber-physical system. *IEEE Systems Journal*, PP(99):1–12, 2017.