Prevent: A Predictive Run-Time Verification Framework Using Statistical Learning

Reza Babaee(✉), Arie Gurfinkel, and Sebastian Fischmeister

Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada
{rbabaeec,arie.gurfinkel,sebastian.fischmeister}@uwaterloo.ca

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 *Prevent*, 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 lookup table 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 that gives the most probable state. We show an empirical evaluation of *Prevent* on a version of randomized dining philosopher and on the QNX Neutrino kernel traces collected from an autopilot software of a hexacopter.

1 Introduction

Run-time Verification (RV) [17] has become a crucial element in developing Cyber-Physical Systems (CPSs) [32,40,42]. 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) [23], 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 accepts (resp. rejects) the prefix. For example, $\varphi_{\mathsf{F}} : \Diamond e$ (resp. $\varphi_{\mathsf{G}} : \Box\neg e$) is satisfied (resp. not satisfied) on any infinite paths with the prefix $u_1 : \neg e,\neg e, e$. Whenever the monitor is not able to reach a verdict with the given prefix π because π can be extended to satisfy and to violate the property, the monitor outputs *unknown* [3]. For example, the prefix $u_2 : \neg e,\neg e$ can be extended to both a path that satisfies $\varphi_{\mathsf{F}} : \Diamond e$ (e.g., any extension of u_1) and a path that violates φ_{F} (e.g., $(\neg e)^{\omega}$).

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 205–220, 2018.
https://doi.org/10.1007/978-3-319-92970-5_13

The monitor is able to reach a verdict with a finite extension of the prefix, if the property is *monitorable* [12]. In this paper, 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. We use Hidden Markov Models [26] (HMMs) to realize a prediction model of the system with partially observable behaviour.

We 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 is the result of a bounded reachability analysis on the product of an HMM and a 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 [38]. Viterbi algorithm generates the most likely state based on a given observation.

We implemented our approach as a proof-of-concept tool[1], called $\mathcal{P}revent$ (Predictive Runtime Verification Framework), and report on using it in two case studies: the original and a modified version of randomized dining philosophers algorithm, and the QNX Neutrino [24] kernel traces. In summary, we make the following contributions:

- introduce $\mathcal{P}revent$, a predictive runtime verification framework to detect satisfaction/violation of a property based on partial execution,
- methodology for constructing a prediction model, that is, the product of a trained HMM and the DFA specifying good and bad extensions,
- define the prediction error on a partial trace and evaluate the monitor performance using hypothesis testing,
- implement the runtime monitoring algorithm using Viterbi approximation,
- evaluate $\mathcal{P}revent$ with two case studies: a modified version of the randomized dining philosophers problem and the flight control of a hexacopter.

The rest of the paper is structured as follows: in Sect. 2, we give an overview of $\mathcal{P}revent$. In Sects. 4 and 5, we provide the details of, respectively, constructing 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 Sect. 6. Finally, we provide the empirical evaluation of $\mathcal{P}revent$ on two case studies in Sect. 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.

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

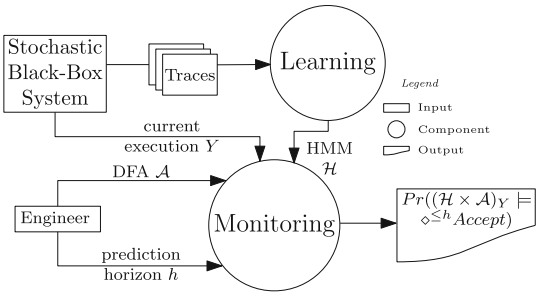


Fig. 1. The overview of $\mathcal{P}revent$ framework.

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, that is represented as the maximum length of the finite extensions. This gives the monitor the ability to detect a property violation before its occurrence.

An overview of $\mathcal{P}revent$ is shown in Fig. 1. The two main components of $\mathcal{P}revent$, *learning* and *monitoring* are described below:

Learning. We use the sample traces to train HMM using Baum-Welch algorithm [26]. The training samples represent an independent and identical distribution (*iid*) over all the execution traces of the system. 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 lookup table. 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 the 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 length of the extensions that are estimated by the monitor. Since $\mathcal{H} \times \mathcal{A}$ has a small size, the probability results of the reachability analysis can be computed off-line for all the states $(\mathcal{H} \times \mathcal{A})$, and 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 the evaluation of the property, and can be obtained empirically from the execution samples.

3 Preliminaries

In this section, we briefly introduce the used definitions and notations.

A probability distribution over a finite set S is a function $P : S \to [0,1]$ 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 values of a random variable X, and use u and w to resp. denote a finite and infinite path.

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 $Pr(y_i \mid X_{1:i}, Y_{1:i}) = Pr(y_i \mid x_i)$ for $i \in [1..\tau]$ (the current observation is conditioned only on the current state), and $Pr(x_i \mid X_{1:i-1}, Y_{1:i-1}) = Pr(x_i \mid x_{i-1})$ 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, $\pi : S \to [0,1]$ is the initial probability distribution over S, $T : S \times S \to [0,1]$ is the transition probability, and $O : S \times \Sigma \to [0,1]$ is the observation probability. We use $\Theta_{\mathcal{H}}$ to denote π, T, and O.*

Discrete-Time Markov Chains (DTMC). We use Discrete-Time Markov Chain (DTMC) for reachability analysis necessary to construct our monitor.

Definition 2 ***(DTMC).*** *A 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, $\pi : S \to [0,1]$ is the initial probability distribution over S, $\mathbf{P} : S \times S \to [0,1]$ is the transition probability, such that for any $s \in S$, $P(s, \cdot)$ is a probability distribution, and $L : S \to \Sigma$ is the labeling function.*

Deterministic Finite Automaton: We use Deterministic Finite Automaton (DFA) to describe the extensions of a prefix.

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, $\delta : Q \times \Sigma \to Q$ 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 ($\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ denotes the language of a 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 value 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.

In the rest of this section, we describe how an HMM is built using standard *Expectation-Maximization* (EM) learning technique [6] (Sect. 4.1), describe the product model of HMM and DFA as a DTMC used to perform the reachability analysis (Sect. 4.2), and, our monitor construction approach (Sect. 4.3).

4.1 Training HMM

We use *Maximum Likelihood Estimation* (MLE) technique [29] 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 $L(\Theta) = \log(\sum_{X_{1:\tau}} Pr(X_{1:\tau}, Y_{1:\tau} \mid \theta))$.

Since the probability distribution over the state sequence $X_{1:\tau}$ is unknown, $L(\Theta)$ does not have a closed form [37], leaving the training techniques to heuristics such as EM. One well-known EM technique for training an HMM is *Baum-Welch* algorithm [26] (BWA), where the training alternates between estimating the distribution over the hidden state variable, $Q : X \rightarrow [0,1]$, with some fixed choice for Θ (*Expectation*), and maximizing the log-likelihood to estimate the values of Θ by fixing Q (*Maximization*) [28].

The *Expectation* phase in BWA computes $Pr(X_t = s \mid Y, \Theta)$ and $Pr(X_t = s, X_{t+1} = s' \mid Y, \Theta)$ for $s, s' \in S$ through *forward-backward* algorithm [26]. *Maximization* is performed on a lower bound of $L(\Theta)$ using Jensen's inequality: $L(\Theta) \geq Q(X) \log Pr(X_{1:\tau}, Y_{1:\tau} \mid \Theta) - Q(X) \log Q(X)$.

Since the second term is independent of Θ [28], only the first term is maximized in each iteration: $\Theta^{(k)} = \text{argmax}_\Theta Q(X) \log Pr(X_{1:\tau}, Y_{1:\tau} \mid \Theta^{(k-1)})$.

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

$$\pi^*(s) = Pr(X_1 = s \mid Y, \Theta) \qquad T^*(s,s') = \frac{\sum_{t=1}^{\tau} Pr(X_t = s, X_{t+1} = s' \mid Y, \Theta)}{\sum_{t=1}^{T} Pr(X_t = s \mid Y, \Theta)}$$

$$O^*(s,o) = \frac{\sum_{t=1}^{\tau} \#(Y_t = o) \cdot Pr(X_t = s \mid Y, \Theta)}{\sum_{t=1}^{T} Pr(X_t = s \mid Y, \Theta)}$$

BWA is essentially a gradient-decent approach, thus its outcome is highly sensitive to the initial values of Θ [37].

We use the Bayesian Information Criterion (BIC) [7] 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 over-fitting: $BIC(\mathcal{H}) = log(n)|\Theta| - 2L(\Theta)$, where $|\Theta| = |S|^2 + |S||\Sigma|$, and n is the size of training sample.

4.2 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 predict expected value of the given property. The expansion of

u is based on a DFA that specifies good and bad extensions of u. The monitor maintains the configurations of both the DFA and the trained HMM by creating the product of the two models [39,41]:

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:*

$$\pi'(s,q,o) = \begin{cases} \pi(s) & \text{if } q \in q_I \\ 0 & \text{otherwise} \end{cases} \qquad L(s,q,o) = \begin{cases} \{Accept\} & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}$$

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

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 $\sigma_0\sigma_1 \cdots \sigma_t$ be the extension of a finite path that ends at the state σ ($\sigma_0 = \sigma \in S'$), such that $L(\sigma_t) = Accept$ in the product model $\mathcal{M}$ ($\sigma_i = (s_i, q_i, o_i)$ is the composite state of the product model $\mathcal{M}$, for all $0 \leq i \leq t$). The monitor's output is $Pr(\sigma_0\sigma_1 \cdots \sigma_t)$, $t \leq h$, which is computed by performing the following reachability analysis on $\mathcal{M}$ [1]: $Pr(\sigma \models \Diamond^{\leq h} Accept)$.

In order to compute this probability we adopt the transformation from [16]:

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

The transformation (1) allows us to recursively compute $Pr(\sigma \models \Diamond^{\leq h} Accept)$ as follows: $Pr(\sigma \models \Diamond^{\leq h} Accept) = \sum_{\sigma'} \mathbf{P}_{Acc}(\sigma,\sigma') Pr(\sigma' \models \Diamond^{\leq h-1} Accept)$ (2).

This is essentially the *transient probability* for $\sigma_0 \cdots \sigma_h w$ [16], that is, starting from σ_0 the probability of being at state σ_h (i.e., after h steps), such that $L(\sigma_h) = Accept$ ($w \in \Sigma^\omega$ is any infinite extension of the path). The probability measure of $\sigma_0 \cdots \sigma_h w$ is based on the prefix $\sigma_0\sigma_1 \cdots \sigma_h$ and can be written as the joint probability distribution of the hidden state variable and that of the observation.

Computing (2) for all the states at runtime is not practical, due to multiplications of large and typically sparse matrices [16]. Instead, for all $t \leq h$ we compute the probabilities off-line and store them in a table $MT(\sigma,t)$, where $MT(\sigma,t) = Pr(\sigma \models \Diamond^{\leq t} Accept)$. Our monitor, thus, is 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.

```
1  MONITOR(Y, H, A, h)
   inputs : Execution observation Y, HMM H = (S, Σ, π, T, O), DFA A = (Q, Σ,
            δ, q_I, F), prediction horizon h
   output: Pr((H × A)_Y ⊨ ◇^{≤h} Accept)
2  begin
3      Construct the monitor table MT(H,A, Σ, h)
4      foreach s ∈ S do v(s) ← O(s,Y_1)π(s)// Initialize the Viterbi vector
5      i ← 1, t ← h, q ← q_I                  // t is the horizon index
6      forall Y_i ∈ Y  do
7          s ← argmax_s v(s)
8          q ← δ(q,Y_i)
9          output MT((s,q,Y_i), t)            // Output the prediction
10         if q ∈ F or t = 0 then t ← h else t ← t − 1
11         forall s ∈ S  do // Updating the next Viterbi vector
12             v_next(s) ← O(s,Y_{t+1}) max_{s'}(v(s')T(s',s))
13         v ← v_next, i ← i + 1
```

Runtime monitoring procedure using Viterbi approximation.

For an observation sequence $Y = Y_{1:\tau}$, Viterbi algorithm [10,38] 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: $v_t(s) = O(s,Y_t) \max_{s' \in S}(v_{t-1}(s')T(s',s))$ (3).

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 that maintain the values of $v_t(s)$ and $v_{t-1}(s)$ (we call them *Viterbi vectors*).

Procedure MONITOR demonstrates our runtime monitoring algorithm. We assume that the monitor table MT is already constructed as described in Sect. 4 (line **3**). Line **4** initialize the Viterbi vector. The *horizon index* t stores the prediction horizon at each iteration (initialized to h at the beginning – line **5**). Each iteration of the *for* loop in lines **6**–**13** is over one observation in the sequence Y. For each observation Y_i, the configuration (s,q,Y_i) (lines **7**–**8**) combined with t gives us the index to retrieve the probability value in the monitor table (line **9**). If the path is not accepted by the DFA, the monitor shrinks its horizon index by one (t is decremented — line **10**). Each time that the observed path is accepted by the DFA, the horizon index is reset to h (line **10**), 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 is reinitialized to h. At the end, the Viterbi vector is updated for the next iteration in lines **11**–**13** according to (3).

In each monitoring iteration (the loop in lines **6-13**), reading the value from the monitor table MT is constant time. For a trained model with k hidden states, updating the Viterbi vector requires $O(k)$ operations of finding maximums, which can be improved to $lg(k)$ using a Max-Heap. Therefore, each monitoring iteration is of $O(k\, lgk)$ in execution time. The space complexity is mainly bounded by the size of the monitor table and the Viterbi vectors: $O(k\, h)$.

6 Prediction Evaluation

In this section, we first define a lower bound on the prediction error of the monitor on a single trace, and then use two-sided hypothesis testing to evaluate the average prediction performance on a set of testing samples. Finally, we exploit the hypothesis testing results to find an empirical lower bound of the horizon.

6.1 Prediction Error

Let $(o_i \cdots o_{i+\lambda_i(\mathcal{A})})$ be an extension of length $\lambda_i(\mathcal{A})$ at point i that is accepted by a given DFA $\mathcal{A}$, i.e., $(o_i \cdots o_{i+\lambda_i}) \in \mathcal{L}(\mathcal{A})$ (for brevity, we use λ_i instead of $\lambda_i(\mathcal{A})$). Recall that the monitor's output at point i is the probability of all the extensions of 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: $Pr(\sigma_i \models \Diamond^{\leq h} Accept) \geq Pr(\sigma_i \cdots \sigma_{i+\lambda_i} \models Accept) \implies \lambda_i \times Pr(\sigma_i \models \Diamond^{\leq h} Accept) \geq \lambda_i \times Pr(\sigma \cdots \sigma_{i+\lambda_i} \models Accept)$.

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: $\varepsilon_i^{min} = \lambda_i - \hat{\lambda}_i$.

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 $(o_i \cdots o_{i+k}) \in \mathcal{L}(\mathcal{A})$, i.e., $(o_i \cdots o_{i+\lambda_i})$ 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 Monitor. As a result, the value of ε_i^{min} can be computed on-the-fly.

In our implementation, we assume that there exists at least one point $k \leq h$ such that $(o_i \cdots o_{i+k}) \in \mathcal{L}(\mathcal{A})$; otherwise, ε_i^{min} 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 rest 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 lower bound 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 value of Λ for a trace is independent of that value for 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: $\frac{\bar{\lambda} - \bar{\lambda}_M}{\frac{\sqrt{\nu}}{n}} \leq t_{n-1,\alpha}$. Given the mean of the length of the shortest finite extensions in the test sample we can use it to obtain a lower bound for h: $h \geq \bar{\lambda} - t_{n-1,\alpha} \frac{\sqrt{\nu}}{n}$.

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 *Prevent* on two case studies: (1) randomized dining philosophers from PRISM [27], which includes the original algorithm, and a modified version that we introduce specifically for evaluating *Prevent*, (2) 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 Dwyer *et al.* [11]'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

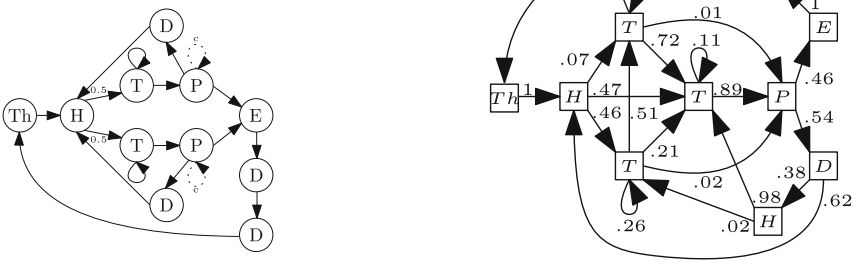
We adapt Rabin and Lehmann [25]'s solution to the dining philosophers problem that has the characteristics of a stochastic system to be trained using HMM. We also present a modification of their algorithm, which represents a generic form of decentralized on-line resource allocation [36], where our monitoring solution can be seen as a component of the *liveness enforcement supervisory* [19].

We consider the classic form of the problem, where philosophers are in a ring topology, and are selected for execution by a fair scheduler. Figure 2a demonstrates a 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 remains at (T) until it is granted access to the fork. The philosopher moves to (E), if the other fork is available; otherwise, the philosopher drops the obtained fork, moves to (D), and eventually transitions back to (H). After the philosopher finishes eating, it drops the forks in an arbitrary order (D), and moves back to (Th). The algorithm is deadlock-free but lockouts are still possible [25].

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) has the probability $1-c$, which is not shown in the figure). This modification enables 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

[2] For simplicity, we remove a self-transition to (Th); however, unlike [9] 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.

means that, instead of going back to (H), the philosopher is more likely to 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 [25] will eventually break, and the algorithm remains deadlock-free. In a distributed real-time system, where each philosopher represents a process with unfixed deadlines, changing the value of c enables the processes to dynamically adjust their waiting time according to their deadlines.



(a) The states of one philosopher. The dotted self-transitions display our modification.

(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 3 philosophers.

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 leads 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 Sect. 4.1). Figure 2b demonstrates the trained HMM of one philosopher that is constructed from the traces of a 5-s 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 as well as other philosophers behaved during training (i.e., resolving non-determinism of the model). For instance, multiple consecutive *try*s in the training sample create several states in the trained HMM,

[3] We tweaked the implementation in https://ti.tuwien.ac.at/tacas2015/ [14].

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

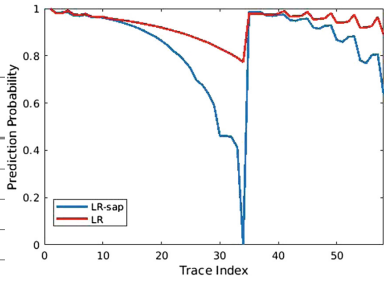


Fig. 3. The comparison of the prediction results from two trained models.

each emitting the symbol (T), but only one has a high probability to transition to (P) and the others model the state where the philosopher can not pick a fork. Finite extensions that we consider in the prediction are based on the following regular expression: $(\neg hungry)^*(hungry(\neg eat)^* eat(\neg hungry)^*)^*$.

Figure 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 has 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). The results in Fig. 3 demonstrate that the model trained on the *bad* extensions (LR-sap) provides an under-approximation of the model that is trained on the *good* traces (LR), thus, producing 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 the 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 $mean(\varepsilon^{min})$. On 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 decreases the error, with the cost of a larger monitor table (MT). The value of $\bar{\lambda}_M$ is influenced by the total number of 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.

7.2 Hexacopter Flight Control

In this section, we apply *Prevent* to detect injected faults from QNX Neutrino's [24] kernel calls. The traces are obtained using QNX `tracelogger` during the flight of a hexacopter[4]. 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`. Figure 4a shows a sub-trace of the kernel call sample from the hexacopter flight control system.

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 CPU 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.

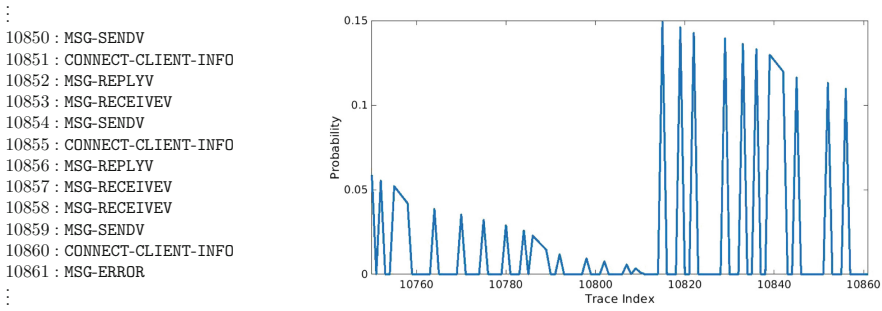
We use SFIHMM [8] on an Intel Xeon 2.40 GHz 128 GB RAM machine with Debian 9.3 to train an HMM from 1-s of the auto-pilot execution, with the intervening process in full effect. The HMM with the minimum BIC has 19 states. The regular expression $(\neg\texttt{MSG\_ERROR})^*\ (\texttt{MSG\_ERROR})\Sigma^*$ is used to generate the finite extensions that contain 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 is partially in effect and started executing in the middle of the flight, is depicted in Fig. 4. The event `MSG_ERROR` is emitted at index 10,861, and the probability of the prefix that contains `MSG_ERROR` within next 50 steps is 0.15 at index 10,815. That means that the monitor predicts the message error with %15 chance, almost 45 steps before its occurrence. The points where the probability is *zero* is because the monitor 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, three consecutive instances of `MSG_RECEIVEV` did not appear 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 several proposals to define semantics of LTL properties on the finite paths [20]; however, to the best of our knowledge, our approach is the first

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



(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.

one in verifying finite paths based on the extensions obtained from a trained HMM. HMMs have been recently used in run-time monitoring of CPSs [2,13,30, 31,33,35,40]. Sistla *et al.* [31] 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-state 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 [30] 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 in [33] to find an upper bound for the timeliness of a monitor can be applied to *Prevent* to find an upper bound for the prediction horizon.

Several works focus on efficiently estimating the internal states of an HMM at runtime using particle filtering [13,35]. 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 [35] 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 *Prevent*.

Learning models for verification is executed on Markov Chain models [18,22]. HMMs are trained in [14] 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 *Prevent*, a predictive run-time monitoring framework for properties with finite regular extensions. The core part of *Prevent* 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* [15].

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 modification to our approach, which would reduce the number of false positives, is to involve a mixture of trained models based on both good and bad traces, and only employing ones that have a higher likelihood to generate the current execution trace.

Lastly, an implementation of $\mathcal{P}revent$ with the application of on-line learning methods (such as state merging or splitting techniques [21,34]) is necessary to apply the framework to the real-world scenarios.

References

1. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Bartocci, E., Grosu, R., Karmarkar, A., Smolka, S.A., Stoller, S.D., Zadok, E., Seyster, J.: Adaptive runtime verification. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 168–182. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35632-2_18
3. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 126–138. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77395-5_11
4. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. **20**(3), 651–674 (2010)
5. Beal, M.J., Ghahramani, Z., Rasmussen, C.E.: The infinite hidden Markov model. In: Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, pp. 577–584. MIT Press, Cambridge (2001)
6. Bilmes, J.A.: 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. Claeskens, G., Hjort, N.L.: Model Selection and Model Averaging. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, Cambridge (2008)
8. DeDeo, S.: Conflict and computation on Wikipedia: a finite-state machine analysis of editor interactions. Futur. Internet **8**(3), 31 (2016)
9. Duflot, M., Fribourg, L., Picaronny, C.: Randomized dining philosophers without fairness assumption. Distrib. Comput. **17**(1), 65–76 (2004)

10. Durbin, R., Eddy, S.R., Krogh, A., Mitchison, G.J.: Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, Cambridge (1998)
11. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 1999 International Conference on Software Engineering, pp. 411–420 (1999)
12. Falcone, Y., Fernandez, J.-C., Mounier, L.: Runtime verification of safety-progress properties. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 40–59. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04694-0_4
13. Kalajdzic, K., Bartocci, E., Smolka, S.A., Stoller, S.D., Grosu, R.: Runtime verification with particle filtering. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 149–166. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1_9
14. Kalajdzic, K., Jegourel, C., Lukina, A., Bartocci, E., Legay, A., Smolka, S.A., Grosu, R.: Feedback control for statistical model checking of cyber-physical systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 46–61. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_4
15. Katoen, J.-P.: Abstraction of probabilistic systems. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 1–3. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75454-1_1
16. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72522-0_6
17. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. **78**(5), 293–303 (2009)
18. Mao, H., Chen, Y., Jaeger, M., Nielsen, T.D., Larsen, K.G., Nielsen, B.: Learning probabilistic automata for model checking. In: 2011 Eighth International Conference on Quantitative Evaluation of Systems, pp. 111–120, September 2011
19. Moody, J., Antsaklis, P.: Supervisory Control of Discrete Event Systems Using Petri Nets. The International Series on Discrete Event Dynamic Systems. Springer, New York (1998). https://doi.org/10.1007/978-1-4615-5711-1
20. Morgenstern, A., Gesell, M., Schneider, K.: An asymptotically correct finite path semantics for LTL. In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 304–319. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_24
21. Mukherjee, K., Ray, A.: State splitting and merging in probabilistic finite state automata for signal representation and analysis. Sig. Process. **104**, 105–119 (2014)
22. Nouri, A., Raman, B., Bozga, M., Legay, A., Bensalem, S.: Faster statistical model checking by means of abstraction and learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 340–355. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_28
23. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57 (1977)
24. Qnx neutrino rtos. http://blackberry.qnx.com/en/products/neutrino-rtos/neutrino-rtos. Accessed 14 Aug 2017
25. Rabin, M.O., Lehmann, D.: The advantages of free choice: a symmetric and fully distributed solution for the dining philosophers problem. In: Roscoe, A.W. (ed.) A Classical Mind, pp. 333–352. Prentice Hall International (UK) Ltd., Hertfordshire (1994)
26. Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. Proc. IEEE **77**(2), 257–286 (1989)

27. Radomised dining philosophers case study. http://www.prismmodelchecker.org/casestudies/phil.php. Accessed 24 Jan 2018
28. Roweis, S.T., Ghahramani, Z.: A unifying review of linear Gaussian models. Neural Comput. **11**(2), 305–345 (1999)
29. Shalev-Shwartz, S., Ben-David, S.: Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press, Cambridge (2014)
30. Sistla, A.P., Srinivas, A.R.: Monitoring temporal properties of stochastic systems. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 294–308. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78163-9_25
31. Sistla, A.P., Žefran, M., Feng, Y.: Monitorability of stochastic dynamical systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 720–736. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_58
32. Sistla, A.P., Žefran, M., Feng, Y.: Runtime monitoring of stochastic cyber-physical systems with hybrid state. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 276–293. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_21
33. Sistla, A.P., Zefran, M., Feng, Y., Ben, Y.: Timely monitoring of partially observable stochastic systems. In: HSCC, 17th International Conference (Part of CPS Week), pp. 61–70 (2014)
34. Stolcke, A., Omohundro, S.M.: Best-first model merging for hidden Markov model induction. CoRR, abs/cmp-lg/9405017 (1994)
35. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_15
36. Tanenbaum, A.S., van Steen, M.: Distributed Systems - Principles and Paradigms, 2nd edn. Pearson Education, London (2007)
37. Terwijn, S.A.: On the learnability of hidden Markov models. In: Adriaans, P., Fernau, H., van Zaanen, M. (eds.) ICGI 2002. LNCS (LNAI), vol. 2484, pp. 261–268. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45790-9_21
38. Viterbi, A.J.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Trans. Inf. Theory **13**(2), 260–269 (1967)
39. Wilcox, C.M., Williams, B.C.: Runtime verification of stochastic, faulty systems. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 452–459. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_34
40. Yavolovsky, A., Žefran, M., Sistla, A.P.: Decision-theoretic monitoring of cyber-physical systems. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 404–419. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_25
41. Zhang, L., Hermanns, H., Jansen, D.N.: Logic and model checking for hidden Markov models. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 98–112. Springer, Heidelberg (2005). https://doi.org/10.1007/11562436_9
42. Zheng, X., Julien, C., Podorozhny, R., Cassez, F., Rakotoarivelo, T.: Efficient and scalable runtime monitoring for cyber-physical system. IEEE Syst. J. **PP**(99), 1–12 (2017)