Runtime Verification with Particle Filtering

Kenan Kalajdzic[1], Ezio Bartocci[1], Scott A. Smolka[2],
Scott D. Stoller[2], and Radu Grosu[1,2]

[1] Faculty of Informatics, Vienna University of Technology, Austria
[2] Department of Computer Science, Stony Brook University, USA

Abstract. We introduce *Runtime Verification with Particle Filtering* (RVPF), a powerful and versatile method for controlling the tradeoff between uncertainty and overhead in runtime verification. Overhead and accuracy are controlled by adjusting the frequency and duration of *observation gaps*, during which program events are not monitored, and by adjusting the number of particles used in the RVPF algorithm. We succinctly represent the program model, the program monitor, their interaction, and their observations as a dynamic Bayesian network (DBN). Our formulation of RVPF in terms of DBNs is essential for a proper formalization of *peek events*: low-cost observations of parts of the program state, which are performed probabilistically at the end of observation gaps. Peek events provide information that our algorithm uses to reduce the uncertainty in the monitor state after gaps.

We estimate the internal state of the DBN using particle filtering (PF) with sequential importance resampling (SIR). PF uses a collection of conceptual particles (random samples) to estimate the probability distribution for the system's current state: the probability of a state is given by the sum of the importance weights of the particles in that state. After an observed event, each particle chooses a state transition to execute by sampling the DBN's joint transition probability distribution; particles are then redistributed among the states that best predicted the current observation. SIR exploits the DBN structure and the current observation to reduce the variance of the PF and increase its performance.

We experimentally compare the overhead and accuracy of our RVPF algorithm with two previous approaches to runtime verification with state estimation: an exact algorithm based on the forward algorithm for HMMs, and an approximate version of that algorithm, which uses precomputation to reduce runtime overhead. Our results confim RVPF's versatility, showing how it can be used to control the tradeoff between execution time and memory usage while, at the same time, being the most accurate of the three algorithms.

1 Introduction

Runtime verification does not come for free. It introduces runtime overhead, thereby altering the timing-related behavior of the program under scrutiny. In applications with realtime constraints, overhead control may be necessary to reduce overhead to an acceptable level.

In previous work [5], we introduced *Software Monitoring with Controllable Overhead* (SMCO), an overhead-control technique that selectively turns monitoring on and off, such that the use of a short- or long-term overhead budget is maximized and never exceeded. Gaps in monitoring, however, introduce uncertainty in the monitoring results.

To quantify the uncertainty, one can estimate the current state of the program. We developed a framework for this, called *Runtime Verification with State Estimation* (RVSE) [10], in which a hidden Markov model (HMM) is used to succinctly model the program and compute the uncertainty in predictions due to incomplete information.

While monitoring is on, the observed program events drive the transitions of the property checker, modeled as a deterministic finite automaton (DFA). They also provide information used to help correct the state estimates (specifically, state probability distributions) computed from the HMM transition probabilities, by comparing the output probabilities in each state with the observed outputs. When monitoring is off, the transition probabilities in the HMM alone determine the updated state estimate after the gap, and the output probabilities in the HMM drive the transitions of the DFA. Each gap is characterized by a *gap length distribution*, which is a probability distribution for the number of missed observations during that gap.

Our algorithm was based on an optimal state estimation algorithm, known as the forward algorithm, extended to handle gaps. Unfortunately, this algorithm incurs high overhead, especially for longer sequences of gaps, because it involves repeated matrix multiplications using the observation-probability and transition-probability matrices. In our measurements, this was often more than a factor of 10 larger than the overhead of monitoring the events themselves!

To reduce the runtime overhead, we developed a version of the algorithm, which we call *approximate precomputed RVSE* (AP-RVSE), that precomputes the matrix calculations and stores the results in a table [1]. Essentially, AP-RVSE precomputes a potentially infinite graph unfolding, where nodes are labeled with state probability distributions, and edges are labeled with transitions. To ensure the table is finite, we introduced an approximation in the calculations, controlled by an accuracy parameter ϵ: if a newly computed matrix differs from the matrix on an existing node by at most ϵ according to the L^1-norm, then we re-use the existing node instead of creating a new one. With this algorithm, the runtime overhead is low, independent of the desired accuracy, but higher accuracy requires larger tables, and the memory requirements could become problematic. Also, if the set of gap length distributions that may appear in an execution is not known in advance, precomputation is infeasible.

This paper introduces an alternative approach, called *Runtime Verification with Particle Filtering* (RVPF), to control the balance among runtime overhead, memory usage, and prediction accuracy. In particle filtering (PF) [7], the probability distribution of states is approximated by the proportion of particles in each state. The particle filtering process works in three recurring steps. First, the particles are advanced to their successor states by sampling from the HMM's transition probability distribution. Second, each particle is assigned a weight corresponding to the output probability of the observed program event. Third, the particles are resampled according to the normalized weights from the second step; this has the effect of redistributing the particles so that they provide a better prediction of the program events.

To reduce the variance of PF, we exploit the knowledge of the current program event and the particular structure of the DBN and employ a variant of PF known as *sequential* *importance resampling* (SIR). The resampling step (which is a performance bottleneck) does not have to be performed in each round, and the particles are advanced to their successor states by sampling from the HMM's transition probability distribution *conditioned by* the current observation. While this conditional probability distribution cannot be computed in general, it can be computed for HMMs.

To handle gaps, we extend PF in a manner that is consistent with the one we devised for the forward algorithm: as long as gaps are the only observations, the particles are advanced to their successor states by sampling from the HMM's transition probability distribution conditioned on the most probable output event. Such output events are chosen by sampling from the output probability distribution of the HMM conditioned on the previous HMM state. These events are used to drive the DFA transitions.

In contrast to our previous work [10,1], we model the HMM, the DFA, and their composition in a more elegant and succinct way as a dynamic Bayesian network (DBN). This allows us to properly formalize a new kind of event, called *peek events*, which are inexpensive observations of part of the program state. In many applications, program states and monitor states are correlated, and hence peek events can be used to narrow down the possible states of the monitor DFA. We use peek events at the end of monitoring gaps to refocus the HMM and DFA states. Our combination of these two kinds of observations, program events and peek events, is akin to *sensor fusion* in robotics.

Adjusting the number of particles used by RVPF provides a versatile way to tune the memory requirements, runtime overhead, and prediction accuracy. With larger numbers of gaps, the particles get more widely dispersed in the state space, and more particles are needed to cover all of the interesting states. To evaluate the performance and accuracy of RVPF, we implemented it along with our previous two algorithms in C and compared them through experiments based on the benchmarks used in [1]. Our results confirm RVPF's versatility. Specifically, we demonstrate in Section 6 that, with the right choice of the number of particles, RVPF consumes 80–100 times less memory than AP-RVSE while being twice as fast as RVSE, and the most accurate of the three algorithms.

The rest of the paper is organized as follows. Section 2 provides background. Sections 3 and 4 define the runtime verification problem we are addressing and system model, respectively. Section 5 presents the RVPF algorithm. Section 6 describes our evaluation methodology and the results of our experiments. Section 7 discusses related work. Section 8 offers concluding remarks and directions for future work.

2 Background

This section provides background information on Bayesian networks, dynamic Bayesian networks, particle filtering, and runtime verification with state estimation.

A *Bayesian network* is a directed acyclic graph in which each node corresponds to a (discrete or continuous) random variable. An edge from node X to node Y indicates that X has a direct influence on Y, and X is called a *parent* of Y. Let B be a Bayesian network over variables $X_1, \ldots, X_n$. Each X_i has a conditional probability distribution $P(X_i \mid \mathit{Parents}(X_i))$ that quantifies the influence of the parents on the node [7].

The meaning of B is a joint distribution over its variables. Let $P(x_1, \ldots, x_n)$ abbreviate $P(X_1 = x_1 \wedge \cdots \wedge X_n = x_n)$, i.e., the conjunction of particular assignments to each variable. Then $P(x_1, \ldots, x_n) = \prod P(x_i \mid \mathit{parents}(X_i))$, where $\mathit{parents}(X_i)$ denotes the values of $\mathit{Parents}(X_i)$ that appear in $x_1, \ldots, x_n$.

A *dynamic Bayesian Network* (DBN) is a Bayesian network that relates random variables to each other over adjacent time steps. Moreover, some variables are observable, and some are not. Let $\mathbf{X_t}$ denote the set of *state variables* at time t. State variables are assumed to be unobservable. Let $\mathbf{O_t}$ denote the set of *observable variables* at time t. The observation at time t is $\mathbf{O_t} = \mathbf{o_t}$ for some set of values $\mathbf{o_t}$.

A *Hidden Markov Model* (HMM) is a special kind of DBN; specifically, an HMM is a DBN with a single state variable and a single observable variable. We refer to a value of the observable variable of an HMM as an *observable action*.

To construct a DBN, one must specify the *prior distribution* $\mathbf{P}(\mathbf{X}_0)$, capturing the initial state distribution; the *transition model* $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1})$, capturing the dependency of the next state on the current state; and the *observation model* $\mathbf{P}(\mathbf{O}_t \mid \mathbf{X}_t)$, encoding the dependency of the observation on the current state. The transition and observation models are represented as a Bayesian network.

Particle filtering (PF) is a sequential Monte Carlo method that can be used to perform state estimation in a Bayesian network [7]. PF can be used to estimate the state probability distribution $\mathbf{P}(\mathbf{X}_t)$, given an observation sequence $o_{1:t}$. In one of the most commonly used forms of particle filtering, known as *sequential importance resampling* (SIR), a population of N_p particles is first created and assigned initial states by sampling from the prior distribution $\mathbf{P}(\mathbf{X}_0)$. A three-step update cycle is then repeated for each time step: (i) each particle is propagated forward by sampling the new state value $\mathbf{x}_t$ given the previous state $\mathbf{x}_{t-1}$ of the particle, based on the transition model $\mathbf{P}(\mathbf{X}_t \mid \mathbf{x}_{t-1})$; (ii) each particle is weighted by the probability it assigns to the new evidence, $\mathbf{P}(\mathbf{o}_t \mid \mathbf{x}_t)$; (iii) the population is *resampled*, i.e., a new population of N_p (unweighted) particles is created, where each new particle is selected from the current population, and the probability that a particular particle is selected is proportional to its weight. Resampling focuses the particles on the high-probability regions of the state space, by probabilistically discarding particles with low weight and duplicating particles with high weight.

One can reduce the variance of PF by using evidence $\mathbf{o}_t$ in the first step of the update cycle by sampling the next state $\mathbf{x}_t$ from the conditional probability distribution $\mathbf{P}(\mathbf{X}_t \mid \mathbf{x}_{t-1}, \mathbf{o}_t)$. As we show in Section 5, this probability distribution can be computed as $P(x_t \,|\, x_{t-1}, o_t) = P(x_t \,|\, x_{t-1}) \cdot P(o_t \,|\, x_t) / P(o_t \,|\, x_{t-1})$. Precomputing $P(o_t \,|\, x_{t-1})$ is possible if the HMM transition probabilities and observation probabilities are given explicitly. By reducing the variance, the resampling frequency (which is a considerable performance bottleneck) can also be reduced.

PF approximates $\mathbf{P}(\mathbf{x}_t \mid \mathbf{o}_{1:t})$, the probability of state $\mathbf{x}_t$ after observation sequence $\mathbf{o}_{1:t}$, by $\frac{1}{N_p} \sum_{i=1}^{N(\mathbf{x}_t | \mathbf{o}_{1:t})} w_i$, where $N(\mathbf{x}_t \mid \mathbf{o}_{1:t})$ is the number of particles in state $\mathbf{x}_t$ after processing observations $\mathbf{o}_1, \ldots, \mathbf{o}_t$ and w_i are the weights of the individual particles which are in state x_t.

Runtime Verification with State Estimation (RVSE) [10] is an algorithm for runtime verification in the presence of observation gaps. In RVSE, a Hidden Markov Model of the monitored program is constructed, monitored event sequences are treated as observation sequences of the HMM, and an extension of the optimal *forward algorithm* for HMM state estimation [7] is used to estimate the state of the HMM and the monitor DFA. We extended the forward algorithm to handle observation gaps, by using the HMM to estimate the unobserved states and events. The time complexity of the RVSE algorithm for a single observation is $O(N_h^2 \cdot N_d)$ for a non-gap event and $O(N_h^2 \cdot N_d^2)$ for a gap event, where N_h and N_d are the numbers of states of the HMM and the DFA, respectively. The *approximately precomputed RVSE* (AP-RVSE) algorithm, described briefly in Section 1, significantly reduces the runtime overhead of RVSE by precomputing and storing the results of the matrix calculations performed by RVSE [1].

3 Problem Statement

The problem statement is based closely on [10]. A problem instance is defined by an HMM H modeling the monitored system, an observation sequence $o_{1:T}$, and a temporal property ϕ over sequences of actions of the monitored system.

The observation sequence contains events that are occurrences of actions performed by the monitored system. In addition, it may contain the symbol $gap(L)$ denoting a possible gap whose length is drawn from a length distribution L, a probability distribution over the natural numbers: $L(\ell)$ is the probability that the gap has length ℓ. In the rest of this paper, we consider a simpler definition of gaps, with gap symbols of form $gap(\ell)$, where the length ℓ of each gap is encoded in the trace.

The HMM H models the monitored system. The HMM need not be an exact model of the system; it simply embodies the available information about the system's behavior. It can be learned automatically from complete traces using standard learning algorithms [7]. Let S_H denote the set of states of the HMM, i.e., the set of possible values of its state variable.

The property ϕ is represented by a DFA $M = \langle S_M, m_{init}, A, \delta, F\rangle$, consisting of a set S_M of states, an initial state m_{init}, an alphabet A, a transition relation δ, and a set F of final (also called "accepting") states. The alphabet A is a subset of the observable actions of the HMM; actions not in A leave the DFA's state unchanged. [1]

The goal is to compute $P(\phi \mid o_{1:T})$, that is, the probability that the system's behavior satisfies ϕ, given observation sequence $o_{1:T}$. This probability is computed from the probability distribution on composite states, where a *composite state* (x, s) is a pair containing an HMM state x and a DFA state s. Specifically,

$$P(\phi \mid o_{1:t}) = \sum_{x_t \in S_H, s_t \in F} P(x_t, s_t \mid o_{1:t}) / \sum_{x_t \in S_H, s_t \in S_M} P(x_t, s_t \mid o_{1:t})$$

where $P(x_t, s_t \mid o_{1:t})$ is the probability that the HMM is in state x_t and the DFA is in state s_t after observation sequence $o_{1:t}$.

4 System Model

The composition of the HMM H modeling the monitored system and the monitor DFA M defines a DBN D representing the entire system. This DBN is illustrated in Figure 1. It shows dependencies among the state variables and observation variables during the t'th time step as well as the dependencies of the state variables X_t and S_t from the

[1] In Section 5 we use a different, HMM-like notation for the DFA.

previous states X_{t-1} and S_{t-1}, respectively. These relationships hold for consecutive observations, without gaps.

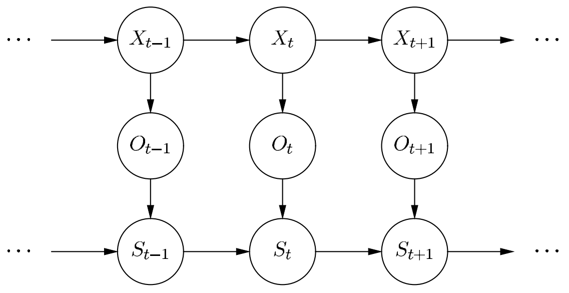

Fig. 1. DBN D composed from the HMM H and the monitor DFA M. X_t and O_t denote the state and observation variables of H at time t, respectively, and S_t denotes the state variables of M, at time t. Note that O_t is also M's input.

4.1 Peek Operations

When a gap occurs, the missing observations cause uncertainty in the state of the DFA. Our algorithm performs a peek operation, which is a lightweight procedure that inspects a part of the monitored system's state immediately after a gap, and can be regarded as an event which is used to reduce the uncertainty in the state of the DFA. Which part of the program state is considered during a peek operation depends on the particular problem and is built into the definition of the procedure that implements the peek operation.

Specifically, peek events are useful for applications in which certain DFA states are known to be inconsistent with certain program states. In such situations, the probabilities associated with composite states containing DFA states which are inconsistent with the partial program state provided by the peek operation can be zeroed, after which the probabilities associated with other composite states are renormalized so that they sum to 1. The additional dependencies between the variables are represented by the DBN in Figure 2.

Because our algorithm uses peek events to reduce uncertainty in the DFA state, we characterize the result of a peek operation q_t by a probability distribution $P(Q_t \mid S_t)$, which is the probability that a peek operation returns Q_t given that the DFA is in state S_t. Using Bayes' rule, after a peek operation that returns q_t after a gap, the probability that the DFA is in state s_t is $P(s_t \mid q_t) = \alpha P(q_t \mid s_t) P(s_t)$, where α is a constant factor used for normalization, and $P(s_t)$ is the probability that the DFA is in state s_t after processing the gap and before processing the peek event.

We do not directly use peek events to reduce uncertainty in the state of the HMM, because generally we do not know a correspondence between concrete program states (provided by peek events) and states of the HMM. This is because the HMM is typically an abstract model learned automatically from traces. However, if such a correspondence is known, then peek events can be used to reduce uncertainty in the state of the HMM, in the same way they are used to reduce uncertainty in the state of the DFA.

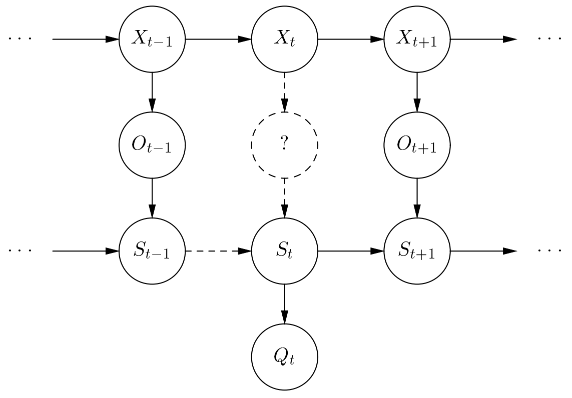

Fig. 2. DBN D composed from the HMM H and the monitor DFA M, when observation o_t is missing due to a gap, and peek event q_t provides information about possible states of the DFA at time t

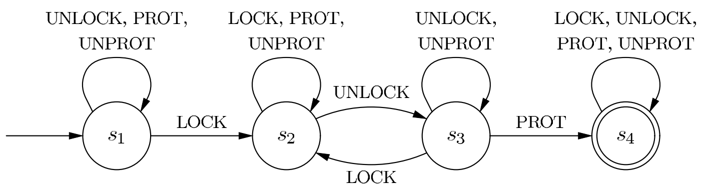

Fig. 3. DFA used to detect violations of a locking discipline

4.2 Running Example

Consider a monitor for a locking discipline for a structure type S in a program. The structure type S contains a lock field (i.e., a field that refers to a lock), protected fields, and unprotected fields. There is a monitor instance for each combination of a thread and a structure of type S. The monitor checks that, when the thread accesses a protected field of the instance of S, the thread holds the lock associated with the instance. The DFA is shown in Figure 3; the parameterization by a thread and a structure is implicit.

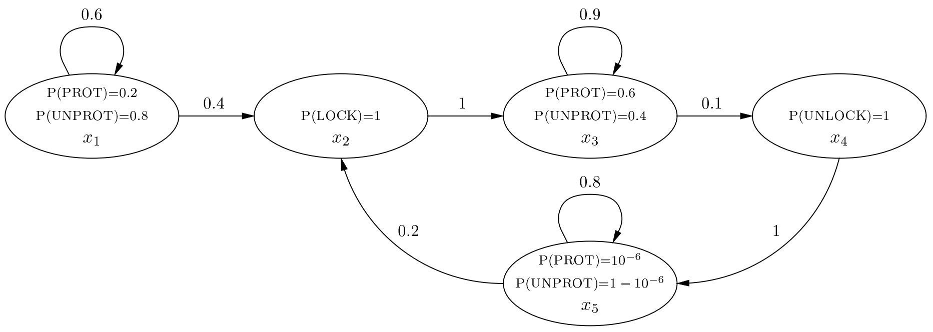

Fig. 4. Graphical representation of the transition and observation probability distributions of an HMM model of a system that usually follows the locking discipline

The alphabet contains four types of events: LOCK, UNLOCK, PROT (representing an access to a protected field) and UNPROT (representing an access to an unprotected field). The states of the monitor have the following interpretation: s_1 – initial state, s_2 – lock is held, s_3 – lock is not held, s_4 – error state (i.e., violation of locking discipline has been detected).

In general, after a gap, the joint probability distribution $P(X_t, S_t)$ may contain non-zero probabilities for all composite states, reflecting uncertainty in the current state of the DFA. The monitor can, however, quickly peek at the state of the lock to check whether it is held by the associated thread. If so, the DFA can only be in states s_2 or s_4, so probabilities of composite states containing s_1 or s_3 can be set to zero. If not, the DFA can only be in states s_1, s_3, or s_4, so the probabilities of composite states containing s_2 can be set to zero. For example, some sample entries in the probability distribution for peek events are $P(s_2 \mid \text{held}) = 1$ and $P(s_2 \mid \text{notHeld}) = 0$.

Figure 4 shows in a graphical way the transition and observation probability distributions of an HMM model of a system that usually follows this locking discipline but has a small chance of violating it.

5 RVPF Algorithm

This section describes our RUNTIMEVERIFICATIONPARTICLEFILTERING (RVPF) algorithm, which performs approximate state estimation based on particle filtering. Like the original RVSE algorithm [10], RVPF estimates the probability that the system is in a composite state (x_t, s_t) at time t. Let $(x_t^{(i)}, s_t^{(i)})$ denote the state, also called the "position", of the i'th particle at time t.

5.1 The Precomputation Phase

In Line 1 of the RVPF algorithm, whose pseudo code is given on page 157, the probabilities $P(O_t \mid X_{t-1})$ and $P(X_t \mid X_{t-1}, O_t)$ are precomputed so they can be accessed quickly by the rest of the algorithm. The exact details of the precomputation are shown in algorithm PRECOMPUTEPROBABILITIES.

Algorithm PRECOMPUTEPROBABILITIES

```
Input:  System Model HMM $H = (X, O, P(X_t \mid X_{t-1}), P(O_t \mid X_t), P(X_0))$
Output: $P(O_t \mid X_{t-1})$, $P(X_t \mid X_{t-1}, O_t)$
1 $P(O_t \mid X_{t-1}) = P(X_t \mid X_{t-1}) P(O_t \mid X_t)$
2 for $i = 1$ to $|\mathrm{dom}(X)|$ do
3     for $j = 1$ to $|\mathrm{dom}(X)|$ do
4         for $k = 1$ to $|\mathrm{dom}(O)|$ do
5             $P(X_t = x_i \mid X_{t-1} = x_j, O_t = o_k) =$
                  $P(X_t = x_i \mid X_{t-1} = x_j) \cdot P(O_t = o_k \mid X_t = x_i) \,/\, P(O_t = o_k \mid X_{t-1} = x_j)$
6         end
7     end
8 end
9 return $[\, P(O_t \mid X_{t-1}),\ P(X_t \mid X_{t-1}, O_t) \,]$
```

Algorithm RUNTIMEVERIFICATIONPARTICLEFILTERING

```
Input:  System Model HMM $H = (X, O, P(X_t \mid X_{t-1}), P(O_t \mid X_t), P(X_0))$,
        Monitor DFA $M = (S, Q, P(S_t \mid S_{t-1}), P(Q_t \mid S_t), P(S_0), F)$,
        Program Events $\boldsymbol{o}_{1:T}$, Peek Events $\boldsymbol{q}_{1:T}$, Number of particles $N_p$
Output: Joint probability distribution $P(X_T, S_T \mid \boldsymbol{o}_{1:T}, \boldsymbol{q}_{1:T})$ after seeing $\boldsymbol{o}_{1:T}$ and $\boldsymbol{q}_{1:T}$
1  $[\,P(O_t \mid X_{t-1}),\ P(X_t \mid X_{t-1}, O_t)\,]$ = PRECOMPUTEPROBABILITIES($H$)
2  $(\boldsymbol{x}_0, \boldsymbol{s}_0, \boldsymbol{w}_0)$ = INITIALIZEPARTICLEDISTRIBUTION($P(X_0)$, $P(S_0)$, $N_p$)
3  for $t = 1$ to $T$ do
4      if $o_t \neq gap$ then
5          for $i = 1$ to $N_p$ do
6              SAMPLE $x_t^{(i)}$ FROM $P(X_t \mid x_{t-1}^{(i)}, o_t)$
7              SAMPLE $s_t^{(i)}$ FROM $P(S_t \mid s_{t-1}^{(i)}, o_t)$
8              $w_t^{(i)} = w_{t-1}^{(i)} \cdot P(o_t^{(i)} \mid x_{t-1}^{(i)})$
9          end
10     else
11         $\ell$ = LENGTH OF GAP
12         $(\boldsymbol{x}_{t-1}, \boldsymbol{s}_{t-1}, \boldsymbol{w}_{t-1})$ = RESAMPLE($\boldsymbol{x}_{t-1}, \boldsymbol{s}_{t-1}, \boldsymbol{w}_{t-1}$)
13         for $i = 1$ to $N_p$ do
14             $(x'_0, s'_0, w'_0) = (x_{t-1}^{(i)}, s_{t-1}^{(i)}, w_{t-1}^{(i)})$
15             for $k = 1$ to $\ell$ do
16                 SAMPLE $o'_k$ FROM $P(O'_k \mid x'_{k-1})$
17                 SAMPLE $x'_k$ FROM $P(X'_k \mid x'_{k-1}, o'_k)$
18                 SAMPLE $s'_k$ FROM $P(S'_k \mid s'_{k-1}, o'_k)$
19                 $w'_k = w'_{k-1} \cdot P(o'_k \mid x'_{k-1})$
20             end
21             $(x_t^{(i)}, s_t^{(i)}, w_t^{(i)}) = (x'_k, s'_k, w'_k)$
22         end
23         for $i = 1$ to $N_p$ do $w_t^{(i)} = w_t^{(i)} \cdot P(q_t \mid s_t^{(i)})$    /* handling a peek event $q_t$ */
24     end
25     NORMALIZE WEIGHTS $w_t$
26     $m = 0$
27     for $i = 1$ to $N_p$ do $m = m + w_i^2$
28     if $1/m \ll N_p \vee q_t \neq \emptyset$ then $(\boldsymbol{x}_t, \boldsymbol{s}_t, \boldsymbol{w}_t)$ = RESAMPLE($\boldsymbol{x}_t, \boldsymbol{s}_t, \boldsymbol{w}_t$)
29 end
30 INITIALIZE MATRIX $P(X_T, S_T \mid \boldsymbol{o}_{1:T}, \boldsymbol{q}_{1:T})$ WITH ZEROS
31 for $i = 1$ to $N_p$ do $P(x_T^{(i)}, s_T^{(i)} \mid \boldsymbol{o}_{1:T}, \boldsymbol{q}_{1:T}) = P(x_T^{(i)}, s_T^{(i)} \mid \boldsymbol{o}_{1:T}, \boldsymbol{q}_{1:T}) + w_T^{(i)}$
32 return $P(X_T, S_T \mid \boldsymbol{o}_{1:T}, \boldsymbol{q}_{1:T})$
```

On Line 1 of PRECOMPUTEPROBABILITIES, the matrix $P(O_t \mid X_{t-1})$ is obtained through a straightforward matrix multiplication of $P(X_t \mid X_{t-1})$ and $P(O_t \mid X_t)$. This is followed by the construction of the 3D-array $P(X_t \mid X_{t-1}, O_t)$ in Lines 2–8.

$P(X_t \mid X_{t-1}, O_t)$ can be best thought of as an array of transition probability matrices $P(X_t \mid X_{t-1})$, one for each observation symbol o_t. This layout makes it possible for the RVPF algorithm to choose the appropriate transition probability distribution depending on the observation symbol generated by the HMM.

5.2 Initial Particle Distribution

The function InitializeParticleDistribution, which is invoked on Line 2 of the RVPF algorithm, distributes N_p particles in the state space based on the initial probability distributions $P(X_0)$ and $P(S_0)$ of the HMM and DFA, respectively.

In the code for this function, variable $D_{i,j}$ holds the number of particles in HMM state x_i and DFA state s_j. The rationale for using $\lceil N_p \cdot P(x_0^{(i)}) \cdot P(s_0^{(j)}) \rceil$ on Line 3 is to guarantee that every state with a non-zero initial probability will contain at least one particle. The code in Lines 1–5 is guaranteed to generate at least N_p particles. If the number of generated particles exceeds N_p, the number is reduced in Lines 6–9 by removing individual particles from the richest states.

Algorithm InitializeParticleDistribution

Input: Initial probability distributions $P(X_0)$ and $P(S_0)$ of the HMM and DFA, respectively, Number of particles N_p

Output: Initial positions $\boldsymbol{x}_0, \boldsymbol{s}_0$ and weights $\boldsymbol{w}_0$ of particles

```
 1  for $i = 1$ to $|\mathrm{dom}(X_0)|$ do
 2      for $j = 1$ to $|\mathrm{dom}(S_0)|$ do
 3          $D_{i,j} = \lceil N_p \cdot P(x_0^{(i)}) \cdot P(s_0^{(j)}) \rceil$
 4      end
 5  end
 6  while $\sum_{i=1}^{|\mathrm{dom}(X_0)|} \sum_{j=1}^{|\mathrm{dom}(S_0)|} D_{i,j} > N_p$ do
 7      FIND $a, b$ FOR WHICH $D_{a,b} = \max(D)$
 8      $D_{a,b} = D_{a,b} - 1$
 9  end
10  $n = 1$
11  for $i = 1$ to $|\mathrm{dom}(X_0)|$ do
12      for $j = 1$ to $|\mathrm{dom}(S_0)|$ do
13          for $k = 1$ to $D_{i,j}$ do
14              $(x_0^{(n)}, s_0^{(n)}, w_0^{(n)}) = (x_i, s_j, 1/N_p)$
15              $n = n + 1$
16          end
17      end
18  end
19  return $(\boldsymbol{x}_0, \boldsymbol{s}_0, \boldsymbol{w}_0)$
```

5.3 Deriving the Optimal Importance Density Function

The simplest form of the SIR particle filter, known as the *bootstrap filter*, uses the transition prior $P(X_t \mid x_{t-1})$ as the importance density function, i.e., the probability distribution from which new particle positions are drawn. Subsequent weight calculations are performed based on the observation probabilities $P(o_t \mid x_t)$, and the particles are then moved to the interesting regions of the state space through resampling. This approach, however, gives poor results in our setting.

The probability distributions of our learned HMMs often have large transition probabilities of x_{t-1} associated with small observation probabilities of x_t, and small transition probabilities of x_{t-1} associated with large observation probabilities of x_t. As a consequence, if the observation o_t corresponds to a low probability transition in x_{t-1}, drawing particles from $P(X_t \mid x_{t-1})$ moves all particles in the "wrong" direction (i.e., contrary to the information provided by the observation), and resampling will have a hard time to move them back to the interesting states.

The solution is to draw new particle positions from an importance density function that takes the observation o_t into account. It has been shown in [3] that the target distribution $P(X_t \mid x_{t-1}, o_t)$ minimizes the variance of importance weights $\boldsymbol{w}_t$ conditioned on $\boldsymbol{x}_{1:t-1}$ and $\boldsymbol{o}_{1:t}$. In practice, it is often difficult to sample from $P(X_t \mid x_{t-1}, o_t)$. Fortunately, in our case, it is possible to obtain $P(X_t \mid X_{t-1}, o_t)$ in closed form [4], which leads to an optimal filter.

5.4 The Forward Step

The loop in Lines 3–29 of the RVPF algorithm estimates the state of the system after each observation. Lines 5–9 handle the regular case in which an observation o_t is available. Lines 11–23 handle gaps.

Handling Program Events. If an observation o_t is available, each particle currently in (x_{t-1}, s_{t-1}) is moved first to (x_t, s_{t-1}) by sampling from $P(X_t \mid x_{t-1}, o_t)$ in Line 6. In next step, the particle is moved to (x_t, s_t) by sampling from $P(S_t \mid s_{t-1}, o_t)$. Note that $P(S_t \mid s_{t-1}, o_t)$ is a conditional probability table which corresponds to the DFA transition function δ. Therefore, the sampling step in Line 7 is guaranteed to return $s_t = \delta(s_{t-1}, o_t)$. Subsequently, in Line 8, the importance weight of each particle is updated by multiplying its current weight with the value from the precomputed matrix $P(O_t \mid X_{t-1})$, where $O_t = o_t$ and $X_{t-1} = x_t$. If the number of particles with significant weights becomes too low (which is estimated in Lines 26–28), the particle positions are resampled in Line 28, based on the weight distribution $\boldsymbol{w}$. This concentrates the particles in the more probable regions of the state space.

Handling Gaps. Upon encountering a gap of length ℓ in the trace, the RVPF algorithm moves each particle, from current state (x_{t-1}, s_{t-1}), ℓ steps forward to state $(x_{t+\ell-1}, s_{t+\ell-1})$, following the probability distributions in the HMM. A single step consists of: sampling an observation o_t from $P(O_t \mid x_{t-1})$ in Line 16; sampling next HMM state x_t from $P(X_t \mid x_{t-1}, o_t)$ in Line 17; sampling next DFA state s_t from $P(S_t \mid s_{t-1}, o_t)$ in Line 18 (which, again, corresponds to advancing the DFA using $s_t = \delta(s_{t-1}, o_t)$); and updating the particle weight on Line 19 using the same equation as on Line 8.

Handling Peek Events. Peek events help correct the movement errors introduced by using the HMM model during gaps. After each gap, a peek operation inspects a variable or a set of variables in the program state and returns an observation q_t. On Line 23, each particle i is weighted by $P(q_t \mid s_t^{(i)})$, the probability DFA state $s_t^{(i)}$ assigns evidence q_t. Particles with impossible DFA states are assigned a weight of zero, and particles with possible DFA states are assigned a weight of 1. The resampling in Line 28 redistributes all particles across the possible DFA states.

5.5 Resampling

Resampling plays a crucial role in maintaining the diversity among particles. Resampling relies on drawing particles from their corresponding weight distribution. To do so, Lines 1–5 of algorithm RESAMPLE compute a distribution containing prefix sums of particle weights: C_i is the sum of weights $w_1, w_2, ..., w_i$. In each iteration of the loop in Lines 6–9, new particle positions are drawn by sampling from C.

Algorithm RESAMPLE

```
Input:  Particle positions and weights $(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{w})$
Output: Particle positions and weights after resampling $(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{s}}, \tilde{\boldsymbol{w}})$

 1  $C_0 = 0$
 2  $N_p = \dim \boldsymbol{x}$
 3  for $i = 1$ to $N_p$ do
 4      $C_i = C_{i-1} + w^{(i)}$
 5  end
 6  for $i = 1$ to $N_p$ do
 7      SAMPLE PARTICLE INDEX $k$ FROM $C$
 8      $(\tilde{x}^{(i)}, \tilde{s}^{(i)}, \tilde{w}^{(i)}) = (x^{(k)}, s^{(k)}, 1/N_p)$
 9  end
10  return $(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{s}}, \tilde{\boldsymbol{w}})$
```

5.6 Calculating the Probability Distribution

After T time steps, the probability distribution $P(X_T, S_T \mid \boldsymbol{o}_{1:T}, \boldsymbol{q}_{1:T})$ is estimated on Lines 30–31 by summing the weights of particles in each state.

6 Evaluation

In this section, we evaluate the performance of the RVPF algorithm by comparing it to the RVSE and AP-RVSE algorithms. We conducted multiple experiments focusing on three important factors: *execution time*, *memory usage*, and state-estimation *accuracy*. All our experiments were carried out under Fedora Linux 17 on a computer with 4GB of RAM and a quad-core Intel® Core$^{\text{TM}}$ i5-2500 CPU running at 3.3GHz. For these experiments, we adapted the existing implementation of AP-RVSE and created new implementations of RVSE and RVPF, reusing relevant parts from the code for AP-RVSE. All three programs are written in C.

The *micro-benchmark* is a multi-threaded application developed for the purpose of experimental evaluation of the AP-RVSE algorithm [1]. It consists of five threads concurrently accessing 100 objects. Each thread can perform four possible operations on any of the objects: LOCK, UNLOCK, PROT, and UNPROT. Threads choose which of these operations to execute according to the HMM in Figure 4. The DFA in Figure 3 is used to check for proper access to the protected fields of each object.

To offer a fair comparison, all three algorithms were evaluated on the same set of micro-benchmark-generated traces (event sequences), which contain gaps of varying frequency (measured as the percentage of trace elements that are gap symbols) and length.

Moreover, each of the algorithms performed state estimation using the same HMM that was used to generate the traces in the first place (i.e., the HMM of Figure 4). In this way, we eliminated the imprecision that might have occurred as a result of re-learning an HMM from the traces, thereby giving each algorithm the opportunity to perform state estimation as accurately as it can. The fact that we used a predefined HMM to drive the micro-benchmark from which we collected the traces allowed us to also log the current state of the HMM and the DFA along with each emitted observation symbol.

6.1 Execution Time and Memory Usage

We conducted experiments aimed at understanding how the number of particles affects the execution time and memory usage of the RVPF algorithm, and how RVPF compares to RVSE and AP-RVSE in terms of these performance measures. For all our experiments, we used both the original HMM (Figure 4) and an additional 10-state HMM learned from the micro-benchmark-generated traces using the Baum-Welch algorithm.

One of our first tests was to measure the execution time of AP-RVSE for different gap lengths (GLs) and gap frequencies (GFs). As expected, in all cases, AP-RVSE always had nearly the same execution time, and was faster than RVPF and RVSE. This was true even when there were no gaps and only two particles were used by RVPF.

The speed of AP-RVSE, however, comes at a price of high memory usage, which is several orders of magnitude higher than that of RVSE and RVPF.

Table 1. Memory consumption in bytes of RVSE, AP-RVPF (with accuracy parameter $\epsilon = 0.1$) and RVPF (for 150 and 350 particles)

Algorithm	RVSE	AP-RVSE	RVPF ($N_p = 150$)	RVPF ($N_p = 350$)
Original 5-state HMM	480	361,240	3,380	6,580
Learned 10-state HMM	960	764,560	5,960	9,160

As Table 1 shows, RVSE uses a relatively small amount of memory, only for storing the HMM and DFA matrices. For RVPF, the amount of required memory is a linear function of the number of particles N_p and was measured to be $16 \cdot N_p + 980$ bytes in case of the original 5-state HMM and $16 \cdot N_p + 3560$ bytes in case of the learned 10-state HMM. In case of the original HMM, with 150 particles RVPF requires around 100 times less memory than AP-RVSE. For the learned HMM and 350 particles, the memory consumption of RVPF is still around 80 times lower than that of AP-RVSE.

We also compared the speed of RVPF to the speed of RVSE. Instead of reporting absolute execution times, we used the execution time of RVSE as the basis for the comparison and determined the number of particles for which RVPF runs exactly as fast as RVSE. For varying GFs and GLs 1-3, we first measured the execution time of RVSE. We then measured the execution time of RVPF with an increasing number of particles until we found the number of particles for which RVPF is exactly as fast as RVSE.

Figure 5 shows that the execution time of RVPF relative to RVSE improves monotonically with respect to the GF, leveling off and reaching a maximum value at a GF of

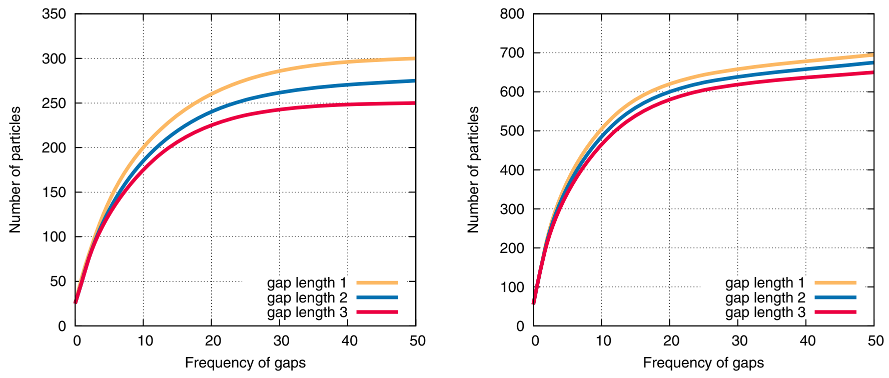

Fig. 5. The number of particles for which RVPF is exactly as fast as RVSE, measured for different GFs and GLs. The figure on the left shows the results for the original 5-state HMM. The figure on the right shows the results for the learned 10-state HMM.

50%. These results also provide a useful guide for choosing the number of particles that maximizes RVPF's accuracy while maintaining its performance advantage over RVSE.

Figure 5 justifies our choice of 150 and 350 particles in Table 1. Namely, with 150 particles RVPF outperforms RVSE for all GLs from 10% to 50% in case of the original 5-state HMM. The same is true for 350 particles when the learned 10-state HMM is used instead.

6.2 Accuracy of State Estimation

Since we recorded the HMM and DFA states in our traces, we can use these values to determine the accuracy of each algorithm's state estimation. We consider first the DFA state. Figure 6 contains our results for estimating the probability for DFA state s_2. The gray line in the graphs serves as a reference value, showing exactly when the DFA was in state s_2. These results are for the worst-case scenario in which there is a gap after each observation symbol (GF = 50%).

The number of particles used for RVPF in obtaining these results was determined as follows. To guarantee that RVPF would always be about twice as fast as RVSE, a significant speed-up, we used the results of Figure 5 to choose the number of particles for RVPF to be half of the value for which it matched RVSE's execution time.

Although we performed state estimation for each DFA state, for presentation purposes, we show only the results for the estimation of DFA state s_2. Even though we are usually interested in state s_4 of the DFA, which is the error state, this state has a very low probability of being reached. The estimated probability of s_4 is therefore almost always zero and rises very slowly. Also, state s_4 is a trap-state, meaning that once entered, the DFA will remain in the state forever. These considerations make s_4 less suitable for measuring accuracy of state estimation. In contrast, s_2 is entered and exited frequently and is thus much more suitable for measuring accuracy of state estimation.

The effect of a 50% GF can be seen in Figure 6 as a form of jitter in the graphs for all three algorithms. Each available observation symbol helps the algorithms increase their certainty, whereas each gap introduces uncertainty. The repeated alternation between visible symbols and gap symbols thus causes the estimated probability to oscillate.

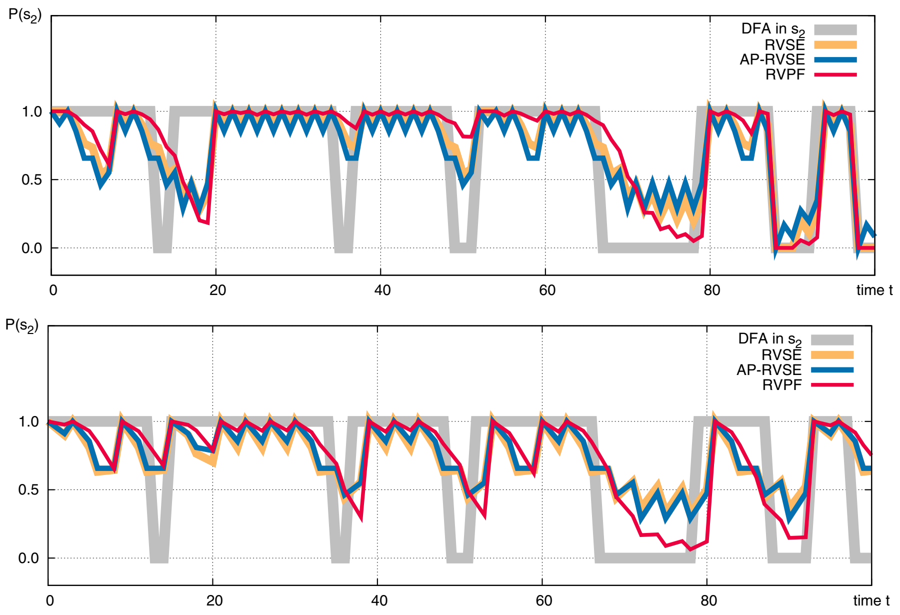

Fig. 6. Measuring accuracy of RVSE, AP-RVSE and RVPF in estimating probability of DFA state s_2 for GF = 50% and GL = 1 (top) and GL = 2 (bottom)

Table 2. Accuracy of RVPF, AP-RVPF and RVSE in estimating probability of DFA state s_2 expressed as L^1-norm of the distance between estimated probability and actual probability at 100 consecutive points in the trace

Algorithm	RVSE	AP-RVSE	RVPF
Gap length 1 (Figure 6 (top))	19.9740	22.5312	17.6149
Gap length 2 (Figure 6 (bottom))	27.1269	24.4361	18.2829
Gap length 2 with peek events (Figure 7)	10.6527	10.2417	8.2252

For each algorithm, we also calculated the L^1-norm of the difference between the estimated probability and the actual probability of DFA state s_2, at 100 consecutive points in the trace. The results are summarized in rows 1 and 2 of Table 2. As the table shows, RVPF gives more accurate results than both RVSE and AP-RVSE in both considered cases (gap length 1 and 2). The reason for this lies in the fact that RVSE and AP-RVSE tend to spread their estimates across all of the states, whereas the limited number of samples drives the estimates of RVPF to the most probable parts of the state space. The RVPF curves in Figure 6 thus show much less jitter and follow the reference curve better than those of RVSE and AP-RVSE. The spreading of estimates across the entire state space in case of RVSE significantly reduces its accuracy as the gap length grows. This can be observed by comparing the results for RVSE and AP-RVSE in the upper two rows of Table 2.

6.3 Estimation Accuracy with Peek Events

To show how peek events help correct estimation errors due to gaps, consider again the results of Figure 6 (bottom), where monitoring is turned off two thirds of the time; i.e., each observation symbol is followed by a gap of length 2. Since, in general, none of the algorithms performs well in this case, we repeated the same test, this time allowing each algorithm to perform a peek operation on the lock after each gap. The noticeable improvements are visible in Figure 7 and quantified in row 3 of Table 2. As expected, peeking results in nearly the same accuracy for RVSE and AP-RVSE, with RVPF being more accurate than both of them for the same reasons given in the previous section (i.e. less jitter and concentration of the estimates in most probable parts of the state space).

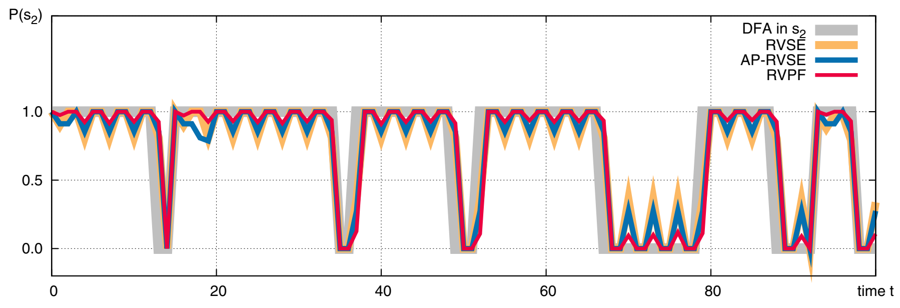

Fig. 7. Estimation of probability of DFA state s_2 for GF = 50% and GL = 2 after correction through peek operations

Peek operations also allow RVPF to correct the estimation of the current state of the HMM. Even though peek events are used only to exclude DFA states from the belief space, the link between a peek observation and the state of the HMM is established through the DBN (Figure 2). This connection allows the peek observation to affect (correct) the estimated probability of the state of the HMM. Figure 8 illustrates the effect of peeking on the estimation of an HMM state by all three algorithms. Recall that we recorded the actual state of the HMM in the traces, depicted in the figures as the gray (reference) curve.

7 Related Work

Particle filtering (PF) has recently been applied to hybrid systems for monitoring and diagnosis purposes, and in particular to estimate the hidden hybrid discrete-continuous state from a set of available measurements [6,2,8,9]. In [6], PF is applied to a class of distributed hybrid systems with autonomous transitions, non-linear system dynamics, and non-Gaussian noise. They demonstrate their approach on a cryogenic propulsion system. In [2], the authors present a PF-based method for discrete-time stochastic hybrid systems, where each particle has two components: a Euclidean component representing the continuous state and a discrete component representing the mode. Their approach combines exact conditional mode probabilities, given the observations, with

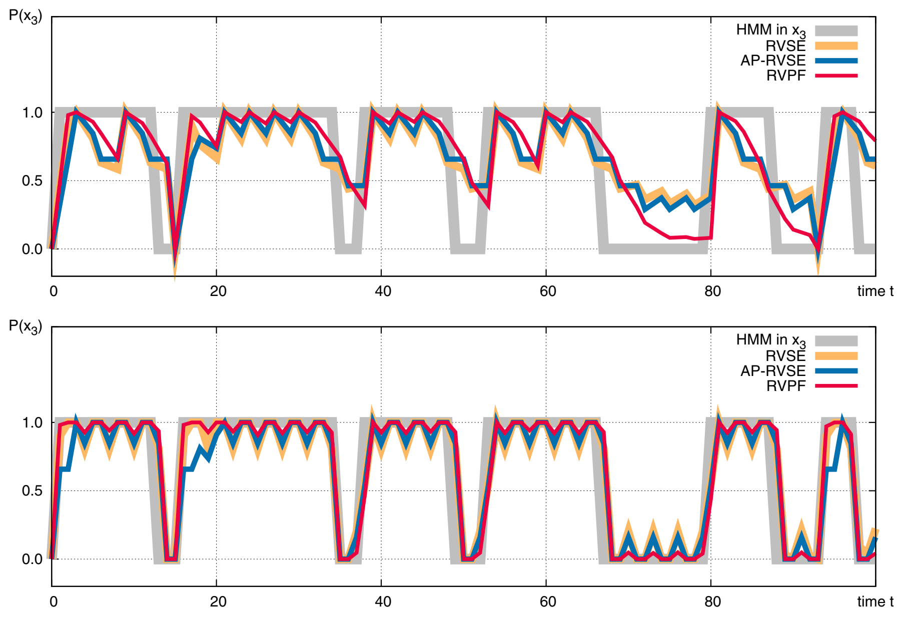

Fig. 8. Estimation of probability of HMM state x_3 with GF = 50% and GL = 1 before and after correction through peek operations

the use of particles to estimate the Euclidean component, showing that this technique works significantly better than standard PF.

Sistla *et al.* use PF to investigate the effectiveness of algorithms for monitorability and strong monitorability of partially observable stochastic systems [8,9]. Familiarity with PF is assumed and no further details, except for the number of particles used, are provided. This application of PF is closest in nature to RVPF but there are significant differences, as witnessed by the contrasting goals of RVPF. In particular, we seek to show how PF can be a highly effective technique for runtime verification, and give a detailed presentation of the RVPF algorithm and its experimental evaluation. Furthermore, we extend PF to handle gaps and peek events. Our experimental results, which compare the accuracy and overhead of RVPF with those of RVSE [10] and approximate precomputed RVSE [1], confirm RVPF's versatility.

The problem we consider—estimating the probability that a safety property is violated by a program execution when monitoring gaps may be present—was introduced in [10]. There an optimal but compute-intensive solution based on the forward algorithm was given. In this paper, we additionally consider *peek events*, which required us to reformulate the problem in terms of DBNs. We also show how to enhance our RVPF algorithm with the *sequential importance resampling* (SIR) strategy using an optimal importance density function, to reduce the variance in state estimation in our setting.

8 Conclusions

This paper introduces RVPF, a versatile method for runtime verification with state estimation in which the balance among runtime overhead, memory usage, and prediction accuracy can be controlled by varying the number of particles RVPF uses for state estimation. Our benchmarking results confirm RVPF's flexibility and its superiority over RVSE and AP-RVSE in terms of state-estimation accuracy.

Although RVPF cannot match the speed of AP-RVSE, its relatively low memory footprint gives it an advantage in the context of embedded systems, where memory resources are limited. Our results also show that RVPF can be configured to outperform RVSE without significantly impacting the accuracy of state estimation.

As future work, we are developing a version of RVPF where the number of particles used for state estimation can vary at runtime. This would allow for dynamic control of the tradeoff involving estimation accuracy, memory consumption, and speed.

Acknowledgements. We thank Justin Seyster for introducing the concept of peek events and developing the micro-benchmarking facility on which our experimental results are based.

References

1. 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)
2. Blom, H.A.P., Bloem, E.: Particle filtering for stochastic hybrid systems. In: Proceedings of 43rd IEEE Conference on Decision and Control, CDC 2004, vol. 3, pp. 3221–3226 (2004)
3. Doucet, A.: Monte Carlo Methods for Bayesian Estimation of Hidden Markov Models. Application to Radiation Signals. Ph.D. Thesis (1997)
4. Doucet, A.: On sequential simulation-based methods for bayesian filtering. Technical Report CUED-F-ENG-TR310, University of Cambridge, Department of Engineering (1998)
5. Huang, X., Seyster, J., Callanan, S., Dixit, K., Grosu, R., Smolka, S.A., Stoller, S.D., Zadok, E.: Software monitoring with controllable overhead. STTT 14(3), 327–347 (2012)
6. Koutsoukos, X.D., Kurien, J., Zhao, F.: Estimation of distributed hybrid systems using particle filtering methods. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 298–313. Springer, Heidelberg (2003)
7. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice-Hall (2010)
8. 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)
9. 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)
10. Stoller, S., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S., Zadok, E.: Runtime verification with state estimation. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012)