# Runtime Verification with State Estimation

**Appears in the proceedings of the 2nd International Conference on Runtime Verification**

**(RV 2011)**

Scott D. Stoller[1], Ezio Bartocci[2], Justin Seyster[1], Radu Grosu[1], Klaus Havelund[3], Scott A. Smolka[1], and Erez Zadok[1]

[1] Department of Computer Science, Stony Brook University, USA
[2] Department of Applied Math and Statistics, Stony Brook University, USA
[3] Jet Propulsion Laboratory, California Institute of Technology, USA

**Abstract.** We introduce the concept of *Runtime Verification with State Estimation* and show how this concept can be applied to estimate the probability that a temporal property is satisfied by a run of a program when monitoring overhead is reduced by sampling. In such situations, there may be *gaps* in the observed program executions, thus making accurate estimation challenging. To deal with the effects of sampling on runtime verification, we view event sequences as observation sequences of a Hidden Markov Model (HMM), use an HMM model of the monitored program to "fill in" sampling-induced gaps in observation sequences, and extend the classic forward algorithm for HMM state estimation (which determines the probability of a state sequence, given an observation sequence) to compute the probability that the property is satisfied by an execution of the program. To validate our approach, we present a case study based on the mission software for a Mars rover. The results of our case study demonstrate high prediction accuracy for the probabilities computed by our algorithm. They also show that our technique is much more accurate than simply evaluating the temporal property on the given observation sequences, ignoring the gaps.

## 1 Introduction

Runtime verification (RV) is the problem of, given a program $P$, execution trace $\tau$ of $P$, and temporal logic formula $\phi$, decide whether $\tau$ satisfies $\phi$. To perform RV, one typically transforms $\phi$ into a *monitor* (a possibly parametrized finite state machine) $M_\phi$ and *instruments* $P$ so that it emits *events* of interest to $M_\phi$. This allows $M_\phi$ to *process* these events and determine whether the event sequence satisfies $\phi$.

RV does not come for free. The *overhead* associated with RV is a measure of how much longer a program takes to execute due to runtime monitoring. If the original program executes in time $R$, and the instrumented program executes in time $R + M$ with monitoring, we say that the monitoring overhead is $\frac{M}{R}$.

Recently, a number of techniques have been developed to mitigate the overhead due to RV $[13, 9, 1, 14, 5]$. Common to these approaches is the use of *event sampling* to reduce overhead. Sampling means that some events are not processed at all, or are processed in a limited (and thus less expensive) manner than other events. A natural question is: *how does sampling affect the results of RV?* This issue has been largely ignored in prior work: the monitor simply reports the result of processing the observed events, without indicating how sampling might have affected the results.

For example, let $\phi$ be the formula $\Box(a \Rightarrow \Diamond c)$ (invariably, $a$ is eventually followed by $c$) and let $\tau$ be the trace $a\,b\,c\,a\,b\,c\,a\,b\,c$. Clearly $\tau$ satisfies $\phi$. Suppose now that $\tau$ is an *incomplete* trace of an execution with implicit gaps due to sampling. Although we cannot decisively say whether the execution satisfies $\phi$ (for example, there could be an unobserved $a$ event after the last $c$ event), we would like to compute a confidence measure that the execution satisfies $\phi$.

In this paper, we introduce the concept of *runtime verification with state estimation* (RVSE), and show how this concept can be applied to estimate the probability that a temporal property is satisfied by a run of a program when monitoring overhead is reduced by sampling. In such situations, there may be *gaps* in observed program executions, making accurate estimation challenging.

The main idea behind our approach is to use a statistical model of the monitored system to "fill in" sampling-induced gaps in event sequences, and then calculate the probability that the property is satisfied. In particular, we appeal to the theory of Hidden Markov Models [17]. An HMM is a Markov model in which the system being modeled is assumed to be a Markov process with unobserved (hidden) states. In a regular Markov model, states are directly visible to the observer, and therefore state transition probabilities are the only required parameters. In an HMM, states cannot be observed; rather, each state has a probability distribution for the possible observations (formally called *observation symbols*). The classic *state estimation* problem for HMMs is to compute the most likely sequence of states that generated a given observation sequence.

The main contributions of this paper are:

- We use HMMs to formalize the RVSE problem as follows. Given an HMM system model $H$, temporal property $\phi$, and observation sequence $O$ (an execution trace that may have gaps due to sampling), compute $\Pr(\phi \mid O, H)$, i.e., the probability that the system's behavior satisfies $\phi$, given $O$ and $H$. Note that we use *Hidden* Markov Models, meaning that the states of the system are hidden from the observer. This is because we intend to use machine learning to learn the HMM from traces that contain only observable actions of the system, not detailed internal states of the system.
- The *forward algorithm* [17] is a classic recursive algorithm for computing the probability that, given an observation sequence $O$, an HMM ended in a particular state. This problem is the so-called *filtering* version of the state estimation problem for HMMs. We present an extension of the forward algorithm for the RVSE problem that computes a similar probability, but in this case for the paired execution of an HMM system model and a monitor automaton for the temporal property $\phi$. We first present a version of the algorithm that does not consider gaps; in this case, the states of the monitor are completely determined by $O$, because the monitor is deterministic.
- We then present an algorithm that handles gaps. We use a special symbol to mark gaps, i.e., points in the observation sequence where unobserved events might have occurred. Gap symbols may be inserted in the trace by the instrumentation when it

temporarily disables monitoring; or, if gaps may occur everywhere, a gap symbol can be inserted at every point in the trace. When the algorithm processes a gap, no observation is available, so the state of the monitor automaton is updated probabilistically based on the current state estimation for the HMM and the observation probability distribution for the HMM. Since the length of a gap (i.e., the number of consecutive unobserved events) might be unknown, we allow the gap length to be characterized by a probability distribution.

– We evaluate our RVSE methodology using a case study based on human operators in a ground station issuing commands to a Mars rover [3]. Sampling of execution traces is simulated using SMCO-style overhead control [14]. Our evaluation demonstrates high prediction accuracy for the probabilities computed by our algorithm. It also shows that our technique is much more accurate than simply evaluating the temporal property on the given observation sequences, ignoring the gaps.

## 2 Related Work

To the best of our knowledge, Runtime Verification with State Estimation has not been studied before, and our HMM-based technique to support the calculation of the conditional probability that a system satisfies a temporal logic formula given a sampled event trace (observation sequence) is new. In this section, we discuss related work on runtime verification of statistical properties and on probabilistic model checking.

Sammapun et al. [18] consider runtime verification of probabilistic properties of the form: given a condition $A$, does the probability that an outcome $B$ occurs fall within a given range? Their technique determines statistically, and with an adequate level of confidence, whether a system satisfies a probabilistic property. Wang et al. [19] apply a similar statistical RV technique, in conjunction with Monte Carlo simulation, to analog and mixed signal designs. Recent work on the runtime verification of probabilistic properties [11, 21] uses acceptance sampling and sequential hypothesis testing to outperform these approaches. In contrast, we perform runtime verification of traditional *non-probabilistic* properties, but in the presence of sampling.

Finkbeiner et al. [10] extend LTL to perform statistical experiments over runtime traces, but they do not consider sampling. For example, their methodology can be used to determine the percentage of positions in a trace at which the trace satisfies a temporal property. This is a different statistic than the conditional probabilities we compute. LarvaStat [7] incrementally computes statistical information about runtime executions, but it, too, does not consider sampling.

Probabilistic model checking [15, 2] can be used to compute the probability that a Markov model, such as a Discrete-Time or Continuous-Time Markov Chain, satisfies a probabilistic temporal logic formula. Zhang et al. [20] extend probabilistic model checking to HMMs, so that the probability that an HMM produces a given sequence of observations can be computed. In contrast, we use HMMs to probabilistically fill in gaps in sampled event traces, enabling us to estimate the probability that a (non-probabilistic) temporal property is satisfied by a trace that contains gaps due to sampling. It is important to note that for filling in the gaps, a considerably less accurate HMM model is acceptable.

## 3 Case Study: A Mars Rover Scenario

We illustrate and evaluate our approach on a software model of a planetary rover mission. The model is written in the SCALA programming language,[4] allowing for fast prototyping. Its architecture, depicted in Figure 1, is representative, in general terms, of actual rover missions, such as the current Mars Science Laboratory[5] (MSL) mission. The scenario we consider consists of a rover operating on the surface of Mars, controlled by commands from ground-based human operators. The rover consists of a collection of instruments (e.g., camera, drill, temperature sensor) performing specialized tasks. For this case study, the rover hosts two generic instruments, *A* and *B*. Furthermore, every event of importance occurring on the rover is recorded in a log, which is maintained on the ground. A ground-based logger module receives and stores such events.

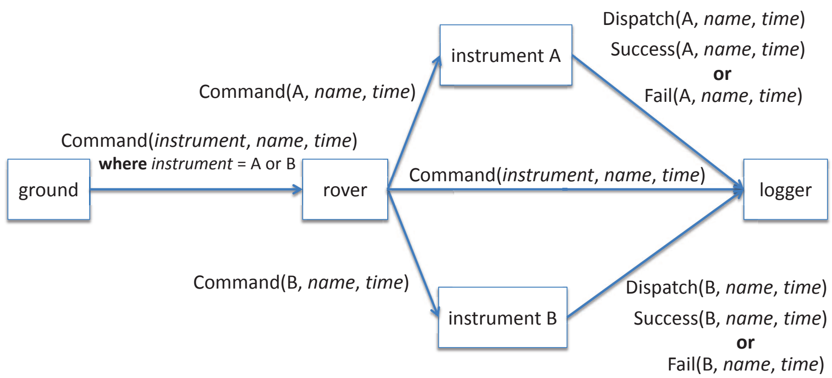


**Fig. 1.** Mission architecture.

| | |
|---|---|
| `Command` (*instrument, name, time*) | commands submitted to rover |
| `Dispatch` (*instrument, name, time*) | dispatch of command from rover to instrument |
| `Success` (*instrument, name, time*) | success of command on instrument |
| `Fail` (*instrument, name, time*) | failure of command on instrument |

**Fig. 2.** Events observed.

We consider four kinds of events, presented in Figure 2 and inspired by the scenario explained by Barringer et al. [3]. Commands are issued from ground to the rover and are characterized by three parameters: instrument id (*A* or *B*), command name, and

[4] `http://www.scala-lang.org`
[5] `http://mars.jpl.nasa.gov/msl`

a time stamp indicating when the event occurred. The other three events have similar parameters. Upon receipt of a command, the rover reports this event to the logger (by sending the command to the logger), and then sends the command to the relevant instrument. The instrument, upon receipt of the command, issues a dispatch event to the logger (recording that it was dispatched to the instrument). The instrument then executes the command. If the execution is successful, a success is reported to the logger. If execution fails, a fail status is reported. It is possible that neither a success nor a fail occur, and that the command is simply lost for some reason. An example log collected during the execution of this system could be: `Command(A, START, 1008)`, `Command(B, RESET, 2303)`, `Success(A, START, 4300)`, `Success(B, RESET, 5430)`.

One aspect of the desired behavior of the rover system is expressed by the requirement: Every `Command`$(i, n, t_1)$ event should eventually be followed by a `Success`$(i, n, t_2)$ event, with no `Fail`$(i, n, t_3)$ event occurring in between.

The above trace satisfies this property. The following trace does not satisfy the property, because the first command fails explicitly, and the second command fails implicitly (neither success nor failure occurs): `Command(A, START, 1008)`, `Command(B, RESET, 2303)`, `Fail(A, START, 4520)`.

This property can be expressed in LTL as follows, where $\Box$ means "always", $\mathcal{U}$ means "until", underscore means "don't care", and the subscript "cs" is mnemonic for "command success".

$$\phi_{\text{cs}} = (\forall\, i : \mathit{Instrument}, n : \mathit{Name}.\; \Box(\texttt{Command}(i, n, \_) \Rightarrow \neg\texttt{Fail}(i, n, \_)\; \mathcal{U}\; \texttt{Success}(i, n, \_))) \tag{1}$$

The property was formulated and checked with TRACECONTRACT [4], a SCALA API for trace analysis supporting parameterized state machines and temporal logic. In TRACECONTRACT, the property is expressed as follows, where SCALA keywords are in bold, TRACECONTRACT features are underlined, and the *hot* state waits for an event that matches the pattern in one of the **case** statements and represents the requirement that such an event eventually occurs:

```
class Contract extends Monitor[Event] {
  require {
    case Command(i,n,_) =>
      hot {
        case Fail(`i`, `n`, _) => error
        case Success(`i`, `n`, _) => ok
      }
  }
}
```

## 4 Background

*Hidden Markov Models.* A Hidden Markov Model (HMM) [17] is a tuple $H = \langle S, A, V, B, \pi \rangle$ containing a set $S$ of states, a transition probability matrix $A$, a set $V$ of observation symbols, an observation probability matrix $B$ (also called "emission probability matrix" or "output probability matrix"), and an initial state distribution $\pi$. The states and observations are indexed (i.e., numbered), so $S$ and $V$ can be written as $S = \{s_1, s_2, \ldots, s_{N_s}\}$ and $V = \{v_1, \ldots, v_{N_o}\}$, where $N_s$ is the number of states, and $N_o$ is the number of observation symbols. Let $\Pr(c_1 \mid c_2)$ denote the probability that $c_1$ holds, given that $c_2$ holds. The transition probability distribution $A$ is an $N_s \times N_s$ matrix indexed by states in both dimensions, such that $A_{i,j} = \Pr(\text{state is } s_j \text{ at time } t+1 \mid \text{state is } s_i \text{ at time } t)$. The observation probability distribution $B$ is an $N_s \times N_o$ matrix indexed by states and observations, such that $B_{i,j} = \Pr(v_j \text{ is observed at time } t \mid \text{state is } s_i \text{ at time } t)$. $\pi_i$ is the probability that the initial state is $s_i$.

An example of an HMM is depicted in the left part of Figure 3. Each state is labeled with observation probabilities in that state; for example, `P(Succ)=.97` in state $\mathtt{s}_3$ means $B_{3,Succ} = 0.97$, i.e., an observation made in state $\mathtt{s}_3$ has probability $0.97$ of observing a `Success` event. Edges are labeled with transition probabilities; for example, `.93` on the edge from $\mathtt{s}_2$ to $\mathtt{s}_3$ means that $A_{2,3} = 0.93$, i.e., in state $\mathtt{s}_2$, the probability that the next transition leads to state $\mathtt{s}_3$ is $0.93$.

An HMM generates observation sequences according to the following five-step procedure [17]. (1) Choose the initial state $q_1$ according to the initial state distribution $\pi$. (2) Set $t = 1$. (3) Choose the $t^{\text{th}}$ observation $O_t$ according to the observation probability distribution in state $q_t$. (4) Choose the next state $q_{t+1}$ according to the transition probability distribution in state $q_t$. (5) Increment $t$ and return to step (3), or stop.

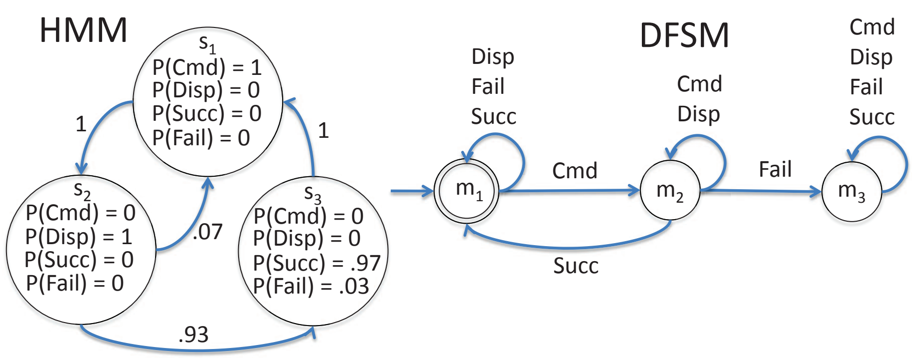


**Fig. 3.** Left: an example of an HMM. The initial state distribution is: $\pi_1 = 1$, $\pi_2 = 0$, $\pi_3 = 0$. Right: $M_{\text{cs}}$, an example of a DFSM. States with a double border are accepting states. In both machines, `Cmd` abbreviates $\mathtt{Command}(\mathtt{i}, \mathtt{n}, \_)$, `Disp` abbreviates $\mathtt{Dispatch}(\mathtt{i}, \mathtt{n}, \_)$, `Succ` abbreviates $\mathtt{Success}(\mathtt{i}, \mathtt{n}, \_)$, and `Fail` abbreviates $\mathtt{Fail}(\mathtt{i}, \mathtt{n}, \_)$.

The *forward algorithm* [17] is a classic algorithm for computing the probability that an HMM ended in a particular state, given an observation sequence $O = \langle O_1, O_2, \ldots, O_T \rangle$. Let $Q = \langle q_1, q_2, \ldots, q_T \rangle$ denote the (unknown) state sequence that the system passed through, i.e., $q_t$ denotes the state of the system when observation $O_t$ is made. Let $\alpha_t(i) = \Pr(O_1, O_2, \ldots, O_t, q_t = s_i \mid H)$, i.e., the probability that the first $t$ observations yield $O_1, O_2, \ldots, O_t$ and that $q_t$ is $s_i$, given the model $H$. To hide the notational clutter from indexing of $V$, we access the $B$ matrix using the traditional notation [17]:

$$b_i(v_k) = B_{i,k} \tag{2}$$

The forward algorithm for computing $\alpha$ is:

$$\alpha_1(j) = \pi_j b_j(O_1) \quad \text{for } 1 \leq j \leq N_s \tag{3}$$

$$\alpha_{t+1}(j) = \left(\textstyle\sum_{i=1..N_s} \alpha_t(i) A_{i,j}\right) b_j(O_{t+1}) \quad \text{for } 1 \leq t \leq T-1 \text{ and } 1 \leq j \leq N_s \tag{4}$$

In the base case, $\alpha_1(j)$ is the joint probability of starting in state $s_j$ and emitting $O_1$. Similarly, the recursive case calculates the joint probability of reaching state $s_j$ and emitting $O_T$. The probability of reaching $s_j$ is calculated by summing over the immediate predecessors $s_i$ of $s_j$; the summand $\alpha_t(i)A_{i,j}$ is the joint probability of reaching $s_i$ while observing $O_1$ through $O_{T-1}$ and then transitioning from $s_i$ to $s_j$. The cost of computing $\alpha$ using these equations is $O(N_s^2 T)$.

*Learning an HMM.* One can obtain an HMM for a system automatically, by learning it from complete traces using standard HMM learning algorithms [17]. These algorithms require the user to specify the desired number of states in the HMM. These algorithms allow (but do not require) the user to provide information about the structure of the HMM, specifically, that certain entries in the transition probability matrix and the observation probability matrix are zero. This information can help the learning algorithm converge more quickly and find globally (instead of locally) optimal solutions. If the temporal property or properties to be monitored are known before the HMM is learned, then the set of observation symbols can be limited to contain only events mentioned in those properties, and the number of states can be chosen just large enough to be able to model the relevant aspects of the system's behavior. Note that we use *Hidden* Markov Models, meaning that the states of the system are hidden from the observer, because we intend to learn $H$ from traces that contain only observable actions of the system, not detailed internal states of the system.

*Deterministic Finite State Machines.* Our algorithm assumes that the temporal property $\phi$ to be monitored is expressed as a parametrized deterministic finite state machine (DFSM). The DFSM could be written directly or obtained by translation from a language such as LTL. A DFSM is a tuple $M = \langle S_M, m_{init}, V, \delta, F\rangle$, where $S_M$ is the set of states, $m_{init}$ in $S_M$ is the initial state, $V$ is the alphabet (also called the set of input symbols), $\delta : S_M \times V \to S_M$ is the transition function, and $F$ is the set of accepting states (also called "final states"). Note that $\delta$ is a total function. A trace $O$ satisfies the property iff it leaves $M$ in an accepting state.

For example, a DFSM $M_{\text{cs}}$ that expresses the property $\phi_{\text{cs}}$ in Equation 1 is depicted in the right part of Figure 3. The `Dispatch` event is not in the alphabet of the TRACECONTRACT property $\phi$ and hence normally would be omitted from the alphabet of the DFSM; we include it in this DFSM for illustrative purposes, so that the alphabets of the HMM and DFSM are the same.

## 5 Algorithm for RVSE

The first subsection defines the problem more formally and presents our algorithm for RVSE. Our algorithm is based on the forward algorithm in Section 4 and hence can be used for on-line or post-mortem analysis. The second subsection describes how we handle parameterized properties.

### 5.1 Problem Statement and Algorithm

A problem instance is defined by an observation sequence $O$, an HMM $H$, and a temporal property $\phi$ over sequences of actions of the monitored system.

The observation sequence $O$ contains events that are occurrences of actions performed by the monitored system. In addition, $O$ may contain the symbol $gap(L)$ denoting a possible gap with an unknown length. The length distribution $L$ is a probability distribution on the natural numbers: $L(\ell)$ is the probability that the gap has length $\ell$.

If no information about the location of gaps is available (and hence no $gap$ events appear in the trace obtained from the runtime monitor), we insert a $gap$ event at the beginning of the trace and after every event in the trace, to indicate that gaps may occur everywhere.

The HMM $H = \langle S, A, V, B, \pi \rangle$ models the monitored system, where $S = \{s_1, \ldots, s_{N_s}\}$ and $V = \{v_1, \ldots, v_{N_o}\}$. Observation symbols of $H$ are observable actions of the monitored system. $H$ need not be an exact model of the system.

The property $\phi$ is represented by a DFSM $M = \langle S_M, m_{init}, V, \delta, F \rangle$. For simplicity, we take the alphabet of $M$ to be the same as the set of observation symbols of $H$. It is easy to allow the alphabet of $M$ to be a subset of the observation symbols of $H$, by modifying the algorithm so that observations of symbols outside the alphabet of $M$ leave $M$ in the same state.

The goal is to compute $\Pr(\phi \mid O, H)$, i.e., the probability that the system's behavior satisfies $\phi$, given observation sequence $O$ and model $H$.

First, we extend the forward algorithm in Section 4 to keep track of the state of $M$. Let $m_t$ denote the state of $M$ immediately after observation $O_t$ is made. Let $\alpha_t(i, m) = \Pr(O_1, O_2, \ldots, O_t, q_t = s_i, m_t = m \mid H)$, i.e., the joint probability that the first $t$ observations yield $O_1, O_2, \ldots, O_t$ and that $q_t$ is $s_i$ and that $m_t$ is $m$, given the model $H$. Let $\mathrm{pred}(n, v)$ be the set of predecessors of $n$ with respect to $v$, i.e., the set of states $m$ such that $M$ transitions from $m$ to $n$ on input $v$. A conditional expression $c\,?\,e_1 : e_2$ equals $e_1$ if $c$ is true, and it equals $e_2$ if $c$ is false. The extended forward algorithm appears below. The main changes are introduction of a conditional expression in equation (6), reflecting that the initial state of $M$ is $m_{init}$, and introduction of a sum over predecessors $m$ of $n$ with respect to $O_{t+1}$ in equation (7), analogous to the existing sum over predecessors $i$ of $j$, so that the sum takes into account all ways of reaching the configuration in which $H$ is in state $s_i$ and $M$ is in state $m$.

$$\mathrm{pred}(n, v) = \{m \in S_M \mid \delta(m, v) = n\} \tag{5}$$

$$\alpha_1(j, n) = (n = \delta(m_{init}, O_1))\,?\,\pi_j b_j(O_1) : 0 \quad \text{for } 1 \le j \le N_s \text{ and } n \in S_M \tag{6}$$

$$\alpha_{t+1}(j,n) = \left( \sum_{\substack{i \in [1..N_s] \\ m \in \mathrm{pred}(n, O_{t+1})}} \alpha_t(i,m) A_{i,j} \right) b_j(O_{t+1}) \tag{7}$$
$$\text{for } 1 \leq t \leq T-1 \text{ and } 1 \leq j \leq N_s \text{ and } n \in S_M$$

Now we extend the algorithm to handle gaps. The result appears in Figure 4. An auxiliary function $p_i$ is used to calculate the probability of transitions of $M$ during gaps. When $H$ is in state $s_i$ and $M$ is in state $m$, $p_i(m,n)$ is the probability that the next observation (i.e., the observation in state $s_i$) causes $M$ to transition to state $n$. Since we do not know which event occurred, we sum over the possibilities, weighting each one with the appropriate observation probability from $B$.

Another auxiliary function $g_\ell$, called the gap transition relation, is used to compute the overall effect of a gap of length $\ell$. Specifically, $g_\ell(i,m,j,n)$ is the probability that, if $H$ is in state $s_i$ and $M$ is in state $m$ and a gap of length $\ell$ occurs, then the $H$ is in state $s_j$ and $M$ is in state $n$ after the gap. The definition of $g_{\ell+1}$ uses a recursive call to $g_\ell$ to determine the probabilities of states reached after a gap of length $\ell$ (these intermediate states are represented by $i'$ and $m'$), and then calculates the effect of the $(\ell+1)^{\text{th}}$ unobserved event as follows: $A_{i',j}$ is the probability that $H$ transitions from state $s_{i'}$ to state $s_j$, and $p_j(m',n)$ is the probability that $M$ transitions to state $n$.

In the definition of $\alpha_1$, for the case $O_1 = \mathit{gap}(L)$, there is a probability $L(0)$ that no gap occurred, in which case $M$ remains in its initial state $m_{init}$ and the probability distribution for states of $H$ remains as $\pi_j$; furthermore, for each $\ell > 0$, there is a probability $L(\ell)$ of a gap of length $\ell$, whose effect is computed by a call to $g_\ell$, and $\pi_i$ is the probability that $H$ is in state $s_i$ at the beginning of the gap.

In the definition of $\alpha_{t+1}$, for the case $O_{t+1} = \mathit{gap}(L)$, there is a probability $L(0)$ that no gap occurred, in which case the state of the HMM and the DFSM remain unchanged, so $\alpha_{t+1}(j,n) = \alpha_t(j,n)$; furthermore, for each $\ell > 0$, there is a probability $L(\ell)$ of a gap of length $\ell$, whose effect is computed by a call to $g_\ell$, and $\alpha_t(i,m)$ is the probability that $H$ is in state $s_i$ and $M$ is in state $m$ at the beginning of the gap.

Although the algorithm involves a potentially infinite sum over $\ell$, typically $L(\ell)$ is non-zero for only a finite number of values of $\ell$, in which case the sum contains only a finite number of non-zero terms. For example, if the system uses lightweight instrumentation to count events during gaps, then the position and length of all gaps are known. In this case, for each gap, $L(\ell)$ is non-zero only for the value of $\ell$ that equals the number of unobserved events (i.e., the gap length). If counts of unobserved events are unavailable (because monitoring is completely disabled during gaps), it is sometimes possible to determine (based on characteristics of the system and how long monitoring was disabled) a threshold such that $L(\ell)$ is non-zero only below that threshold. Even if no such threshold exists, $L(\ell)$ typically approaches 0 as $\ell$ becomes large, so the sum can be approximated by truncating it after an appropriate number of terms.

$$p_i(m,n) = \sum_{v\in V \text{ s.t. } \delta(m,v)=n} b_i(v) \tag{8}$$

$$g_0(i,m,j,n) = (i = j \wedge m = n)\,?\,1:0 \tag{9}$$

$$g_{\ell+1}(i,m,j,n) = \sum_{i'\in[1..N_s],m'\in S_M} g_\ell(i,m,i',m')A_{i',j}p_j(m',n) \tag{10}$$

$$\alpha_1(j,n) = \tag{11}$$
$$\begin{cases} (n = \delta(m_{init},O_1))\,?\,\pi_j b_j(O_1):0 & \text{if } O_1 \neq \mathit{gap}(L) \\ L(0)(n = m_{init}\,?\,\pi_j:0) + \sum_{\ell>0,i\in[1..N_s]} L(\ell)\pi_i g_\ell(i,m_{init},j,n) & \text{if } O_1 = \mathit{gap}(L) \end{cases}$$
$$\text{for } 1 \leq j \leq N_s \text{ and } n \in S_M$$

$$\alpha_{t+1}(j,n) = \begin{cases} \left(\sum_{\substack{i\in[1..N_s] \\ m\in\mathrm{pred}(n,O_{t+1})}} \alpha_t(i,m)A_{i,j}\right) b_j(O_{t+1}) & \text{if } O_{t+1} \neq \mathit{gap}(L) \\ L(0)\alpha_t(j,n) + \sum_{\ell>0} L(\ell) \sum_{\substack{i\in[1..N_s] \\ m\in S_M}} \alpha_t(i,m)g_\ell(i,m,j,n) & \text{if } O_{t+1} = \mathit{gap}(L) \end{cases} \tag{12}$$
$$\text{for } 1 \leq t \leq T-1 \text{ and } 1 \leq j \leq N_s \text{ and } n \in S_M$$

**Fig. 4.** Forward algorithm modified to handle gaps.

### 5.2 Handling Parameterized Temporal Properties

Our approach supports parameterized temporal properties. Specified events trigger creation of a new instance of the parameterized property, and parameters of the trigger event are used as parameters of the property. For example, the property $\phi_{\text{cs}}$ in equation (1), and the corresponding DFSM $M_{\text{cs}}$ in Figure 3, are parameterized by the instrument `i` and the name `n`. The parameters of the DFSM may be used in the definition of the alphabet of the DFSM; in other words, the alphabet is also parameterized. For example, the alphabet of $M_{\text{cs}}$ is $\{\texttt{Command}(i,n,\_),\texttt{Dispatch}(i,n,\_),\texttt{Success}(i,n,\_),\texttt{Fail}(i,n,\_)\}$.

For a parameterized property, we decompose (or "demultiplex") a given trace into a set of subtraces by projecting it onto the alphabet of each instance of the property. The HMM is learned from these subtraces; thus, the HMM represents the slice of the system's overall behavior relevant to a single instance of the property. When learning the HMM, we abstract from the specific values of the parameters in each subtrace, because the values are, of course, different in each subtrace, and we do not aim to learn the distribution of parameter values.

When applying our modified forward algorithm for a parameterized property, we run the algorithm separately for each instance of the property, and use the corresponding subtrace (i.e., the projection of the trace onto the alphabet of that property instance) as the observation sequence $O$.

When projecting a trace containing gaps onto the alphabet of a property instance, it is typically unknown whether the unobserved event or events that occurred during a gap are in that alphabet. This can be reflected by modifying the length distribution parameter of the gap symbol appropriately before inserting the gap in the subtrace for that property instance. Developing a method to modify the length distribution appropriately, based on the nearby events in the trace and the HMM, is future work. Lee et al.'s work on trace slicing [16] might provide a basis for this.

The above approach does not assume any relationship between the property parametrization and the sampling strategy. An alternative approach is to adopt a sampling strategy in which, for each property instance, either all relevant events are observed, or none of them are. For example, when QVM [1] checks properties of Java objects, it selects some objects for checking, monitors all events on those objects, and monitors no events on other objects. With this approach, the property is checked with 100% confidence for the selected objects, but it is not checked at all for other objects. This trade-off might be preferable in some applications but not in others. Also, this property-directed sampling may incur more overhead than property-independent sampling, because it must ensure that all events relevant to the selected property instances are observed.

# 6 Evaluation

## 6.1 Evaluation Methodology

We used the following methodology to evaluate the accuracy of our approach for a given system.

1. Produce a set $T_L$ of traces by monitoring the system without sampling, and learn an HMM $H$ from them.
2. Produce another set $T_E$ of traces by monitoring the system without sampling, and use them for evaluation as follows.
3. Produce a sampled version $\check{O}$ of each trace $O$ in $T_E$. If the system is deterministic, $\check{O}$ can be produced by re-running the system on the same input as for $O$ while using sampling. An alternative approach, applicable regardless of whether the system is deterministic, is to write a program that reads a trace, simulates the effect of sampling, and outputs a sampled version of the trace.
4. For each trace $O$ in $T_E$, apply our algorithm to compute the probability $\Pr(\phi|\check{O}, H)$.
5. Compare the probabilities from the previous step to reality, by partitioning the traces in $T_E$ into "bins" (i.e., sets) based on $\Pr(\phi|\check{O}, H)$, and checking whether the expected fraction of the traces in each set actually satisfy $\phi$. Specifically, using $B + 1$ bins, for $b \in [0..B]$, the set of traces placed in bin $b$ is $T_E(b) = \{O \in T_E \mid b/B \le \Pr(\phi|\check{O}, H) < (b+1)/B\}$. Let $sat_{act}(b)$ denote the fraction of traces in bin $b$ that actually satisfy $\phi$. Based on the results from our algorithm, $sat_{act}(b)$ is expected to be approximately $sat_{est}(b) = \mathrm{average}(\{\Pr(\phi|\check{O}, H) \mid O \in T_E(b)\})$. The subscript "est" is mnemonic for "estimation", i.e., "expected based on state estimation".
6. Quantify the overall inaccuracy as a single number $I$ between 0 and 1, where 0 means perfect accuracy (i.e., no inaccuracy), by summing the differences between

the actual and expected fractions from the previous step for non-empty bins and normalizing appropriately ("ne" is mnemonic for "non-empty"):

$$B_{\mathrm{ne}} = \{b \in [0..B] \mid T_E(b) \neq \emptyset\} \tag{13}$$

$$I = \frac{1}{|B_{\mathrm{ne}}|} \sum_{b \in B_{\mathrm{ne}}} |sat_{act}(b) - sat_{est}(b)|. \tag{14}$$

7. Put this inaccuracy into perspective by comparing it with the inaccuracy of the naive approach that ignores the effect of sampling and simply evaluates the property on sampled traces, ignoring gaps. Specifically, $sat_{naive}(b)$ is the fraction of traces in $T_E(b)$ such that the sampled trace satisfies $\phi$, i.e., $sat_{naive}(b) = |\{O \in T_E(b) \mid \check{O} \models \phi\}|/|T_E(b)|$, and

$$I_{naive} = \frac{1}{B_{\mathrm{ne}}} \sum_{b \in B_{\mathrm{ne}}} |sat_{act}(b) - sat_{naive}(b)|. \tag{15}$$

If the sampling strategy has a parameter that controls how many events are observed, then the inaccuracy $I$ can be graphed as a function of that sampling parameter. For example, SMCO has a parameter $o_t$, the target overhead. We expect the inaccuracy to approach 0 as the fraction of events that are observed approaches 1. Similarly, for a particular trace $O$, $\Pr(\phi|\check{O}, H)$ can be graphed as a function of that sampling parameter; if the trace $O$ satisfies $\phi$, this curve should monotonically increase towards 1 as the fraction of events that are observed approaches 1.

### 6.2 Experiments

We applied the above methodology to the rover case study described in Section 3. The SCALA model was executed to generate 200 traces, each containing 200 issued commands. The average length of the traces is 587 events. To facilitate evaluation of our approach, the model was modified to pseudo-randomly introduce violations of the requirement $\phi_{\mathrm{cs}}$ in Equation 1. Approximately half of the traces satisfy the requirement. In the other half of the traces, the requirement is violated by approximately 30% of the commands; among those commands, approximately half have an explicit `Fail` event, and the other half do not have a `Success` or `Fail` event. We wrote a program that reads a trace, simulates the sampling performed by SMCO with a global controller [14], and then outputs the trace with some events replaced by $gap(L_0)$, where $L_0(0) = 0$, $L_0(1) = 1$, and $L_0(\ell) = 0$ for $\ell > 1$. Note that $gap(L_0)$ represents a definite gap of length 1. The use of a definite gap reflects that the SMCO controller knows when it disables and enables monitoring, and that (in an actual implementation) lightweight instrumentation would be used to count the number of unobserved events when monitoring is (mostly) disabled. With the target overhead that we specified, the SMCO simulator replaced 47% of the events with gaps.

Based on the parameters of the property $\phi_{\mathrm{cs}}$, each sampled trace was decomposed into a separate subtrace for each instrument and command, following the approach in Section 5.2. When decomposing the trace, we assigned each gap to the appropriate subtrace by referring to the original (pre-sampling) trace. Although it is generally unrealistic to assume that the monitor can assign gaps to subtraces with 100% accuracy, this assumption allows us to isolate this source of inaccuracy and defer consideration of it to future work, in which we plan to introduce uncertain gaps into subtraces corresponding to nearby events in the full trace, using the HMM to compute probabilities for the uncertain gaps.

To obtain the HMM $H$, we manually specified the number of states (six) and the structure of the HMM, and then learned the transition probability matrix and observation probability matrix from half of the generated traces. We used the other half of the generated traces for evaluation.

We measured the inaccuracy of our approach using $B = 10$, and obtained $I = 0.0205$. This level of inaccuracy is quite low, considering the severity of the sampling: recall that sampling replaced 47% of the events with gaps. In comparison, the inaccuracy of the naive approach is $I_{naive} = 0.3135$; this is approximately a 15× worse $I$.

## 7 Conclusions and Future Work

This paper introduces the new concept of *Runtime Verification with State Estimation* (RVSE) and shows how this concept can be applied to estimate the probability that a temporal property is satisfied by a run of a system given a sampled execution trace. An initial experimental evaluation of this approach shows encouraging results.

One direction for future work, mentioned in Section 5.2, is to determine the probability that a gap belongs to each subtrace of a parameterized trace, in order to more accurately determine the length distribution parameter for gap events inserted in subtraces. Because the parameters of events in gaps are unknown, it is impossible to directly determine the subtrace to which a gap belongs.

Although our Mars rover case study is based on actual rover software, due to ITAR restrictions, our evaluation used parametrized event traces synthetically produced by a simulator. We plan to conduct additional case studies involving actual traces obtained from publicly available real-world software. Likely target software systems include the GCC compiler suite and the Linux kernel.

Another direction for further study is RVSE of quantitative properties. For example, the goal of integer range analysis [9, 14] is to compute the range (upper and lower bounds) of each integer variable in the program. Performing this kind of analysis on traces with gaps can lead to inaccuracies in the ranges computed, due to unobserved updates to integer variables. In this case, we would like to extend our RVSE algorithm to adjust (improve) the results of the analysis as well as provide a confidence level in the adjusted results. Similar comments apply to other quantitative properties, such as runtime analysis of NAPs (non-access periods) for heap-allocated memory regions [13, 14].

Our broader goal is to use probabilistic models of program behavior, learned from traces, for multiple purposes, including program understanding [6], program visualization [8], and anomaly detection [12] (by checking future runs of the program against the model).

*Acknowledgements.* We would like to thank the anonymous reviewers for their valuable comments. Part of the research described in this publication was carried out at the Jet Propulsion Laboratory (JPL), California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Research supported in part by AFOSR Grant FA9550-09-1-0481, NSF Grants CCF-1018459, CCF-0926190, CNS-0831298, and ONR Grant N00014-07-1-0928.

## References

1. Arnold, M., Vechev, M., Yahav, E.: QVM: An efficient runtime for detecting defects in deployed systems. In: Proc. 23rd ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008). pp. 143–162. ACM (Oct 2008)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
3. Barringer, H., Groce, A., Havelund, K., Smith, M.: Formal analysis of log files. Journal of Aerospace Computing, Information, and Communication 7(11), 365–390 (2010)
4. Barringer, H., Havelund, K.: TraceContract: A Scala DSL for trace analysis. In: Proc. 17th International Symposium on Formal Methods (FM 2011). Lecture Notes in Computer Science, vol. 6664. Springer (2011)
5. Bonakdarpour, B., Navabpour, S., Fischmeister, S.: Sampling-based runtime verication. In: Proc. 17th International Symposium on Formal Methods (FM 2011). Springer (Jun 2011)
6. Buss, E., Henshaw, J.: Experiences in program understanding. In: Proc. Second Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1992). pp. 157–189. IBM Press (1992)
7. Colombo, C., Gauci, A., Pace, G.J.: LarvaStat: Monitoring of statistical properties. In: Proc. First International Conference on Runtime Verification (RV 2010). pp. 480–484. Springer (2010)
8. Diehl, S.: Software Visualization: Visualizing the Structure, Behavior, and Evolution of Software. Springer (2007)
9. Fei, L., Midkiff, S.P.: Artemis: Practical runtime monitoring of applications for execution anomalies. In: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006). pp. 84–95. ACM, Ottawa, Canada (June 2006)
10. Finkbeiner, B., Sankaranarayanan, S., Sipma, H.B.: Collecting statistics over runtime executions. Form. Methods Syst. Des. 27, 253–274 (Nov 2005)
11. Grunske, L.: An effective sequential statistical test for probabilistic monitoring. Information and Software Technology 53, 190–199 (March 2011)
12. Hangal, S., Lam, M.S.: Tracking down software bugs using automatic anomaly detection. In: Proc. 24th International Conference on Software Engineering (ICSE 2002). pp. 291–301. ACM (2002)
13. Hauswirth, M., Chilimbi, T.M.: Low-overhead memory leak detection using adaptive statistical profiling. In: Proc. 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004). pp. 156–164 (Oct 2004)
14. Huang, X., Seyster, J., Callanan, S., Dixit, K., Grosu, R., Smolka, S.A., Stoller, S.D., Zadok, E.: Software monitoring with controllable overhead. International Journal on Software Tools for Technology Transfer (2011)
15. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07). LNCS, vol. 4486, pp. 220–270. Springer (2007)

16. Lee, C., Chen, F., Roşu, G.: Mining parametric specifications. In: Proc. 33rd International Conference on Software Engineering (ICSE 2011). pp. 591–600. ACM (2011)
17. Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE 77(2), 257–286 (1989)
18. Sammapun, U., Lee, I., Sokolsky, O., Regehr, J.: Statistical runtime checking of probabilistic properties. In: Proc. 7th International Workshop on Runtime Verification (RV 2007). Lecture Notes in Computer Science, vol. 4839, pp. 164–175. Springer (2007)
19. Wang, Z., Zaki, M., Tahar, S.: Statistical runtime verification of analog and mixed signal designs. In: Proc. Third International Conference on Signals, Circuits and Systems (SCS 2009). pp. 1–6. IEEE (Nov 2009)
20. Zhang, L., Hermanns, H., Jansen, D.N.: Logic and model checking for hidden Markov models. In: Proc. 25th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2005). Lecture Notes in Computer Science, vol. 3731, pp. 98–112. Springer (2005)
21. Zhang, P., Ki, W., Wan, D., Grunske, L.: Monitoring of probabilistic timed property sequence charts. Software: Practice and Experience 41, 841–866 (Jun 2011)