Adaptive Runtime Verification

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

[1] Department of Computer Science, Stony Brook University, USA
[2] Department of Computer Engineering, Vienna University of Technology

Abstract. We present *Adaptive Runtime Verification* (ARV), a new approach to runtime verification in which overhead control, runtime verification with state estimation, and predictive analysis are synergistically combined. Overhead control maintains the overhead of runtime verification at a specified target level, by enabling and disabling monitoring of events for each monitor instance as needed. In ARV, predictive analysis based on a probabilistic model of the monitored system is used to estimate how likely each monitor instance is to violate a given temporal property in the near future, and these *criticality levels* are fed to the overhead controllers, which allocate a larger fraction of the target overhead to monitor instances with higher criticality, thereby increasing the probability of violation detection. Since overhead control causes the monitor to miss events, we use Runtime Verification with State Estimation (RVSE) to estimate the probability that a property is satisfied by an incompletely monitored run. A key aspect of the ARV framework is a new algorithm for RVSE that performs the calculations in advance, dramatically reducing the runtime overhead of RVSE, at the cost of introducing some approximation error. We demonstrate the utility of ARV on a significant case study involving runtime monitoring of concurrency errors in the Linux kernel.

1 Introduction

In [11], we introduced the concept of *runtime verification with state estimation* (RVSE), and showed how it can be used to estimate the probability that a temporal property is satisfied by a partially or incompletely monitored program run. In such situations, there may be *gaps* in observed program executions, making accurate estimation challenging.

Incomplete monitoring can arise from a variety of sources. For example, in real-time embedded systems, the sensors might have intrinsically limited fidelity, or the scheduler might skip monitoring of internal or external events due to an impending deadline for a higher-priority task. Incomplete monitoring also arises from overhead control frameworks, such as [5], which repeatedly disable and enable monitoring of selected events, to maintain the overall overhead of runtime monitoring at a specified target level. Regardless of the cause, simply ignoring the fact that unmonitored events might have occurred gives poor results.

The main idea behind RVSE is to use a statistical model of the monitored system, in the form of a Hidden Markov Model (HMM), to "fill in" gaps in event sequences. We then use an extended version of the forward algorithm of [7] to calculate the probability

that the property is satisfied. The HMM can be learned automatically from training runs, using standard algorithms [7].

When the cause of incomplete monitoring is overhead control, a delicate interplay exists between RVSE and overhead control, due to the runtime overhead of RVSE itself: the matrix-vector calculations performed by the RVSE algorithm to process an observation symbol—which can be a program event or a gap symbol paired with a discrete probability distribution describing the length of the gap—are expensive. Note that we did not consider this interplay in [11], because the RVSE calculations were performed post-mortem in the experiments described there.

The relationship between RVSE and overhead control can be viewed as an accuracy-overhead tradeoff: the more overhead RVSE consumes processing an observation symbol, with the goal of performing more accurate state estimation, the more events are missed (because less overhead is available). Paradoxically, these extra missed events result in more gap symbols, making accurate state estimation all the more challenging.

This tension between accurate state estimation and overhead control can be understood in terms of Heisenberg's uncertainty principle, which essentially states that the more accurately one measures the position of an electron, the more its velocity is perturbed, and vice versa. In the case of RVSE, we are estimating the position (state) and velocity (execution time) of a "computation particle" (program counter) flowing through an instrumented program.

With these concerns in mind, this paper presents *Adaptive Runtime Verification* (ARV), a new approach to runtime verification in which overhead control, runtime verification with state estimation, and predictive analysis are synergistically combined. In ARV, as depicted in Figure 1, each monitor instance[3] has an associated *criticality level*, which is a measure of how "close" the instance is to violating the property under investigation. As criticality levels of monitor instances rise, so will the fraction of monitoring resources allocated to these instances, thereby increasing the probability of violation detection and concomitant adaptive responses to property violations.

The main contributions of this paper are:

- In ARV, the overhead-control subsystem and the RVSE-enabled monitoring subsystem are coupled together in a feedback control loop: overhead control introduces gaps in event sequences, whose resolution requires HMM-based state estimation (RVSE); state estimation informs overhead control, closing the loop. Up-to-date state estimates enable the overhead-control subsystem to make intelligent, criticality-based decisions about how to allocate the available overhead among monitor instances.
- A key aspect of the ARV framework is a new algorithm for RVSE that performs the calculations offline (in advance), dramatically reducing the runtime overhead of RVSE, at the cost of introducing some approximation error. We analyze the cumulative approximation error incurred by this algorithm.
- To compute the criticality levels of monitor instances, the ARV framework performs reward-based reachability queries over the Discrete Time Markov Chain

[3] A *monitor instance* is a runtime instance of a parameterized monitor. For example, our monitor for concurrency errors in the Linux kernel is parameterized by the id (address) of the structure being monitored.

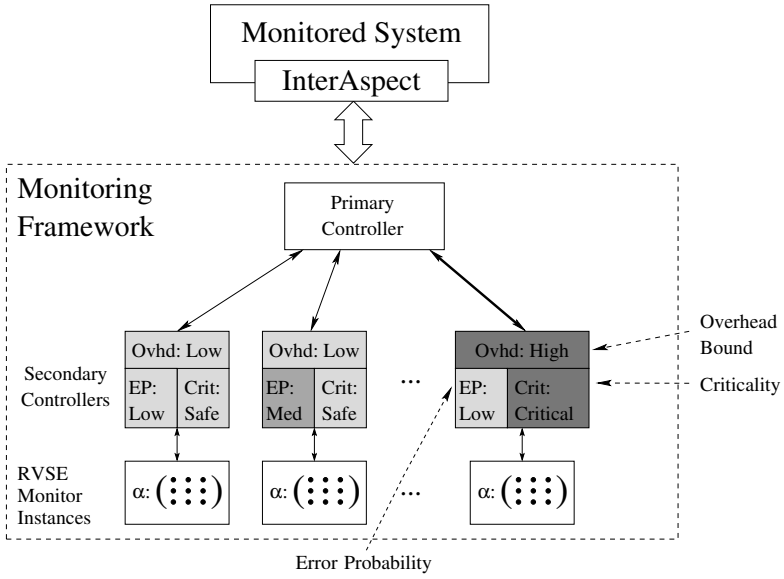

Fig. 1. The Adaptive Runtime Verification Framework

(DTMC) derived from the composition of the HMM model of the monitored program and the monitor, represented as a Deterministic Finite State Machine (DFSM). These queries determine the expected distance to the monitor's error state. These queries are also computed in advance, and the results are stored in a data structure.

– We demonstrate the utility of the ARV approach on a significant case study involving runtime monitoring of concurrency errors in the Linux kernel.

2 Background

Hidden Markov Models (HMMs). An HMM [7] is a tuple $H = \langle S, A, V, B, \pi \rangle$ containing a set S of states, a transition probability distribution A, a set V of observation symbols (also called "outputs"), an observation probability distribution B, and an initial state distribution π. 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 $B_{i,j} = \Pr(v_j \text{ is observed at time } t \mid \text{ state is } s_i \text{ at time } t)$. Following tradition, we define $b_i(v_k) = B_{i,k}$. Prior distribution π_i is the probability that the initial state is s_i.

An example of an HMM is depicted in Figure 3 a). Each state is labeled with observation probabilities in that state; for example, `P(LOCK)` =0.99 in state s_1 means

$B_{1,LOCK} = 0.99$. Edges are labeled with transition probabilities; for example, 0.20 on the edge from s_2 to s_3 means $A_{2,3} = 0.20$.

Learning HMMs. Given a set of traces of a system and a desired number of states of the HMM, it is possible to learn an HMM model of the system using standard algorithms [7]. The main idea behind these algorithms is to maximize the probability that the HMM generates the given traces. In our experiments, we chose an HMM model with three states, used the Baum-Welch learning algorithm [1], and provided the learning algorithm with 1,000 traces as input. Figure 3 a) depicts the transition and observation probability distributions of the resulting HMM model. The related case study (Section 6) provides further details.

Deterministic Finite State Machines (DFSMs). We assume that the temporal property ϕ to be monitored is expressed as a parametrized deterministic finite state machine. 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), $\delta : S_M \times V \rightarrow S_M$ is the transition function, and F is the set of accepting states (also called "final states"). Note that δ is a total function. A trace O satisfies the property iff it leaves M in an accepting state.

RVSE Algorithm. In [11], we extended the forward algorithm to estimate the probability of having encountered an error (equivalent to be in an accepting state) in the case where the observation sequence O contains 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 ℓ.

The Hidden Markov Model $H = \langle S, A, V, B, \pi \rangle$ 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 ϕ 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 ϕ, given observation sequence O and model H. 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. We extend the forward algorithm [7] to compute $\alpha_t(i, m) = \Pr(O_1, O_2, \ldots, O_t, q_t = s_i, m_t = m \mid H)$, 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. We refer to a pair (j, n) of an HMM state and a DFSM state as a *compound state*, and we sometimes refer to α_t as a probability distribution over compound states. The extended algorithm appears in Figure 2. The desired probability $\Pr(\phi \mid O, H)$ is the probability that the DFSM is in an accepting state after observation sequence O, which is $p_{\text{sat}}(\alpha_{|O|+1})$, where $p_{\text{sat}}(\alpha) = \sum_{j \in 1..N_s, n \in F} \alpha(j, n) \;/\; \sum_{j \in 1..N_s, n \in S_M} \alpha(j, n)$. The probability of an error (i.e., a violation of the property) is $p_{\text{err}}(\alpha) = 1 - p_{\text{sat}}(\alpha)$.

$$p_i(m,n) = \sum_{v \in V \text{ s.t. } \delta(m,v)=n} b_i(v) \tag{1}$$

$$g_0(i,m,j,n) = (i = j \wedge m = n)\,?\,1:0 \tag{2}$$

$$g_{\ell+1}(i,m,j,n) = \sum_{i' \in [1..N_s], m' \in S_M} g_\ell(i,m,i',m') A_{i',j} p_j(m',n) \tag{3}$$

$$\alpha_1(j,n) = \tag{4}$$
$$\begin{cases} (n = \delta(m_{init}, O_1))\,?\,\pi_j b_j(O_1) : 0 & \text{if } O_1 \neq gap(L) \\ L(0)(n = m_{init}\,?\,\pi_j : 0) + \sum_{\ell>0, i \in [1..N_s]} L(\ell)\pi_i g_\ell(i, m_{init}, j, n) & \text{if } O_1 = gap(L) \end{cases}$$
$$\text{for } 1 \leq j \leq N_s \text{ and } n \in S_M$$

$$\alpha_{t+1}(j,n) = \begin{cases} \left(\sum_{\substack{i \in [1..N_s] \\ m \in \mathrm{pred}(n, O_{t+1})}} \alpha_t(i,m) A_{i,j} \right) b_j(O_{t+1}) & \text{if } O_{t+1} \neq gap(L) \\ L(0)\alpha_t(j,n) + \sum_{\ell>0} L(\ell) \sum_{\substack{i \in [1..N_s] \\ m \in S_M}} \alpha_t(i,m) g_\ell(i,m,j,n) & \text{if } O_{t+1} = gap(L) \end{cases} \tag{5}$$
$$\text{for } 1 \leq t \leq T-1 \text{ and } 1 \leq j \leq N_s \text{ and } n \in S_M$$

Fig. 2. Forward algorithm for Runtime Verification with State Estimation. $\mathrm{pred}(n, v)$ is the set of predecessors of n with respect to v in the DFSM, i.e., the set of states m such that M transitions from m to n on input v.

3 The ARV Framework: Architecture and Principles

Figure 1 depicts the architecture of the ARV framework. ARV uses InterAspect [8], an aspect-oriented program-instrumentation framework that we developed for the GCC compiler collection, to insert code that intercepts monitored events and sends them to the *monitoring framework*. The monitoring framework maintains a separate RVSE-enabled monitor instance for each monitored object.

Each monitor instance uses the RVSE algorithm in Section 4 to compute its estimate of the composite HMM-DFSM state; specifically, it keeps track of which pre-computed probability distribution over compound states characterizes the current system state.

Each instance uses this probability distribution over compound states to compute the its *error probability* (EP), i.e., the probability that a property violation has occurred, as described in Section 4. Each instance also uses this probability distribution over compound states to compute its *criticality level*, based on the expected number of transitions before a violation occurs, using the predictive analysis of Section 5.

The overhead-control subsystem is structured, as in SMCO [5], as a *cascade controller* comprising one primary controller and a number of secondary controllers, one per monitor instance. The primary controller allocates monitoring resources (overhead), and the secondary controllers enforce the overhead allocation by disabling monitoring

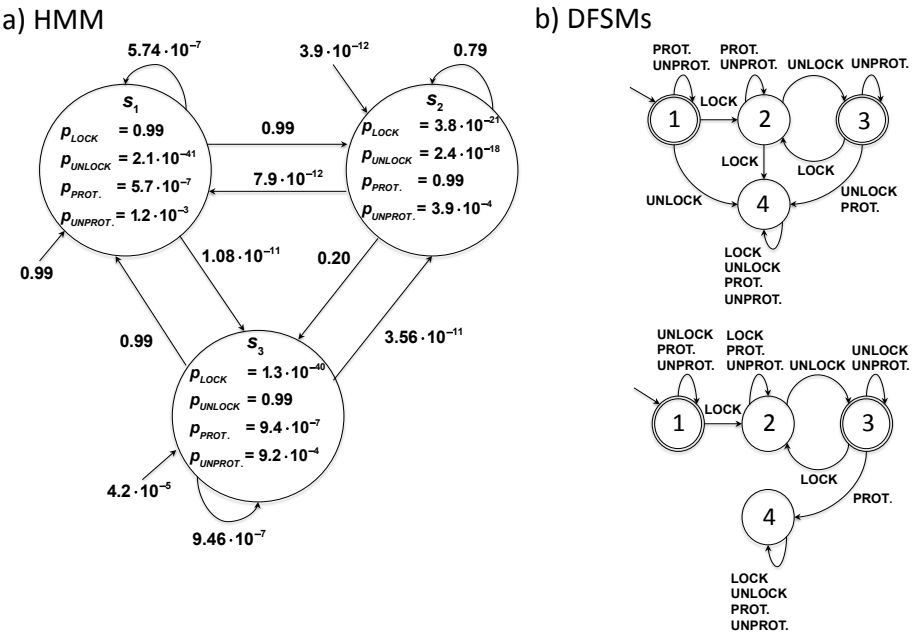

Fig. 3. Left (a): An example of an HMM. Right (b): Two examples of DFSM. States with a double border are accepting states.

when necessary. A key feature of ARV's design is the ability to redistribute overhead so that more critical monitor instances are allowed more monitoring overhead.

4 Pre-computation of RVSE Distributions

Performing the matrix calculations in the RVSE algorithm during monitoring incurs very high overhead. This section describes how to dramatically reduce the overhead by pre-computing compound-state probability distributions α and storing them in a rooted graph. Each edge of the graph is labeled with an observation symbol. At run-time, the algorithm maintains (for each monitor instance) a pointer $curNode$, indicating the node associated with the current state. The probability distribution in the current state is given by the matrix associated with $curNode$. Initially, $curNode$ points to the root node. Upon observing an observation symbol O, the algorithm finds the node n' reachable from $curNode$ by an edge labeled with O, and then assigns n' to $curNode$. Note that this takes constant time, independent of the sizes of the HMM and the monitor.

In general, an unbounded number of probability distributions may be reachable, in which case the graph would be infinite. We introduce an approximation in order to ensure termination. Specifically, we introduce a binary relation $\mathrm{closeEnough}$ on compound-state probability distributions, and during the graph construction, we identify nodes that are close enough.

α_0 = the probability distribution with $\alpha_0(j, m_{init}) = \pi_0(j)$, and $\alpha_0(j, n) = 0$ for $n \neq m_{init}$
$workset = \{\alpha_0\}$
$nodes = \{\alpha_0\}$
while $workset \neq \emptyset$
 $\alpha = workset$.removeOne();
 for each observation symbol O in V
  $\alpha' = \mathrm{normalize}(\mathrm{successor}(\alpha, O))$
  if $\mathrm{dead}(\alpha')$
   continue
  endif
  if $\alpha' \in nodes$
   add an exact edge labeled with O from α to α'
  else if there exists α'' in $nodes$ such that $\mathrm{closeEnough}(\alpha', \alpha'')$
   add an approximate edge labeled with O from α to α'
  else
   add α' to $nodes$ and $workset$
   add an exact edge labeled with O from α to α'
  endif
 endfor
endwhile

Fig. 4. Pseudo-code for graph construction

Pseudo-code for the graph construction appears in Figure 4. $\mathrm{successor}(\alpha, O)$ is the probability distribution obtained using the forward algorithm—specifically, equation (5)—to update compound-state probability distribution α based on observation of observation symbol O. Note that each edge is marked as *exact* or *approximate*, indicating whether it introduces any inaccuracy. $\mathrm{normalize}(\alpha)$ is the probability distribution obtained by computing $\sum_{j,n} \alpha(j, n)$ and then dividing every entry in α by this sum; the resulting matrix α' satisfies $\sum_{j,n} \alpha'(j, n) = 1$. Normalization has two benefits. First, it helps reduce the number of nodes, because normalized matrices are more likely to be equal or close-enough than un-normalized matrices. Second, normalization helps reduce the inaccuracy caused by the use of limited-precision numerical calculations in the implementation (*cf.* [7, Section V.A]), which uses the term "scaling" instead of "normalization"). Normalization is compatible with our original RVSE algorithm—in particular, it does not affect the value calculated for $\Pr(\phi \mid O, H)$—and it provides the second benefit described above in that algorithm, too, so we assume hereafter that the original RVSE algorithm is extended to normalize each matrix α_t.

A state s of a DFSM is *dead* if it is non-accepting and all of its outgoing transitions lead to s. A probability distribution is *dead* if the probabilities of compound states containing dead states of the DFSM sum to 1. The algorithm does not bother to compute successors of dead probability distributions, which always have error probability 1.

We define the close-enough relation by: $\mathrm{closeEnough}(\alpha, \alpha')$ iff $||\alpha - \alpha'||_{\mathrm{sum}} \leq \epsilon$, where ϵ is an implicit parameter of the construction, and $||\alpha||_{\mathrm{sum}} = \sum_{i,j} |\alpha(i, j)|$. Note that, if we regard α as a vector, as is traditional in HMM theory, then this norm is the vector 1-norm.

Termination. We prove termination of the graph construction using the pigeonhole principle. Consider the space of $N_s \times N_m$ matrices with entries in the range $[0..1]$, where $N_m = |S_m|$. Partition this space into cells (hypercubes) with edge length $\epsilon/N_s N_m$. If two matrices α and α' are in the same cell, then the absolute value of the largest element in $\alpha - \alpha'$ is at most $\epsilon/N_s N_m$, and $||\alpha - \alpha'||_{\text{sum}}$ is at most the number of elements times the largest element, so $||\alpha - \alpha'||_{\text{sum}} \leq N_s N_m \epsilon / N_s N_m$, hence $||\alpha - \alpha'||_{\text{sum}} \leq \epsilon$. The contrapositive of this conclusion is: if two matrices satisfy $||\alpha - \alpha'||_{\text{sum}} > \epsilon$, then they are in different cells. Therefore, the number of nodes in the graph is bounded by the number of cells in this grid, which is $(N_s N_m/\epsilon)^{N_s N_m}$. Note that this termination proof applies even if normalization is omitted from the algorithm.

Cumulative Inaccuracy. Use of the closeEnough relation during graph construction introduces inaccuracy. We characterize the inaccuracy by bounding the difference between the probability distribution matrix associated with $curNode$ and the probability distribution matrix that would be computed by the original RVSE algorithm. Let $\alpha'_1, \alpha'_2, \ldots, \alpha'_t$ be the sequence of matrices labeling the nodes visited in the graph, for a given observation sequence O. Let $\alpha_1, \alpha_2, \ldots, \alpha_t$ be sequence of matrices calculated by the RVSE algorithm for the same observation sequence O. The cumulative inaccuracy is expressed as a bound err_t on $||\alpha_t - \alpha'_t||_{\text{sum}}$. First, we consider inaccuracy assuming that the original and new RVSE algorithms do *not* normalize the probability distributions (recall that normalization is not needed to ensure soundness or termination), and we show that the cumulative inaccuracy does not increase along an exact edge and increases by at most ϵ along an approximate edge.

We define err_t inductively. The base case is $err_0 = 0$. For the induction case, we suppose $||\alpha'_t - \alpha_t||_{\text{sum}} \leq err_t$ and define err_{t+1} so that $||\alpha'_{t+1} - \alpha_{t+1}||_{\text{sum}} \leq err_{t+1}$.

If the transition from α'_t to α'_{t+1} traverses an exact edge, then the inaccuracy remains unchanged: $err_{t+1} = err_t$. To prove this, we show that the following inequality holds: $||\text{successor}(\alpha'_t, O_{t+1}) - \text{successor}(\alpha_t, O_{t+1})||_{\text{sum}} \leq err_t$. To prove this, we expand the definition of successor and simplify. There are two cases, depending on whether O_{t+1} is a gap. If O_{t+1} is not a gap,

$$
\begin{aligned}
& \textstyle\sum_{j \in [1..N_s], n \in S_M} |\text{successor}(\alpha'_t, O_{t+1}) - \text{successor}(\alpha_t, O_{t+1})| \\
= & \textstyle\sum_{j \in [1..N_s], n \in S_M} \sum_{i \in [1..N_s], m \in \text{pred}(n, O_{t+1})} |\alpha'_t(i,m) - \alpha_t(i,m)| A_{i,j} b_j(O_{t+1}) \\
& \text{// } M \text{ is deterministic, so each } m \text{ is predecessor of at most one } n \text{ for given } O_{t+1}, \\
& \text{// so for any } f, \textstyle\sum_{n \in S_M, m \in \text{pred}(n, O_{t+1})} f(m) \leq \sum_{m \in S_M} f(m). \\
\leq & \textstyle\sum_{j \in [1..N_s], i \in [1..N_s], m \in S_M} |\alpha'_t(i,m) - \alpha_t(i,m)| A_{i,j} b_j(O_{t+1}) \\
& A \text{ is stochastic, i.e., } \textstyle\sum_{j \in S_M} A_{i,j} = 1, \text{ and } b_j(O_{t+1}) \leq 1 \\
\leq & \textstyle\sum_{i \in [1..N_s], m \in S_M} |\alpha'_t(i,m) - \alpha_t(i,m)| \\
\leq & \; err_t
\end{aligned}
$$

If O_{t+1} is a gap,

$$
\begin{aligned}
&\textstyle\sum_{j\in[1..N_s],n\in S_M} |\mathrm{successor}(\alpha'_t, O_{t+1}) - \mathrm{successor}(\alpha_t, O_{t+1})| \\
&= \textstyle\sum_{j\in[1..N_s],n\in S_M} L(0)|\alpha'_t(j,n) - \alpha_t(j,n)| \\
&\quad + \textstyle\sum_{\ell>0} L(\ell) \sum_{i\in[1..N_s],m\in\mathrm{pred}(n,O_{t+1})} |\alpha'_t(i,m) - \alpha_t(i,m)| g_\ell(i,m,j,n) \\
&\quad \textit{// definition of } err_t \\
&\leq L(0) err_t \\
&\quad + \textstyle\sum_{\ell>0} L(\ell) \sum_{i\in[1..N_s],m\in\mathrm{pred}(n,O_{t+1})} |\alpha'_t(i,m) - \alpha_t(i,m)| \sum_{j\in[1..N_s],n\in S_M} g_\ell(i,m,j,n) \\
&\quad \textit{// } g_\ell(i,m,\cdot,\cdot) \text{ is stochastic, i.e., } \textstyle\sum_{j\in[1..N_s],n\in S_M} g_\ell(i,m,j,n) = 1, \text{ and def. of } err_t \\
&\leq L(0) err_t + \textstyle\sum_{\ell>0} L(\ell) err_t \\
&\quad \textit{// } \textstyle\sum_{\ell\geq 0} L(\ell) = 1 \\
&\leq err_t
\end{aligned}
$$

If the transition from α'_t to α'_{t+1} traverses an approximate edge, then, by definition of closeEnough, the traversal may add ϵ to the cumulative inaccuracy, so $err_{t+1} = err_t + \epsilon$. Note that the same argument used in the case of an exact edge implies that the inaccuracy in err_t is not amplified by traversal of an approximate edge.

Now we consider the effect of normalization on cumulative inaccuracy. We show that normalization does not increase the inaccuracy. Let $\hat{\alpha}'_t$ and $\hat{\alpha}_t$ be the probability distributions computed in step t before normalization; thus, $\alpha'_t = \mathrm{normalize}(\hat{\alpha}'_t)$ and $\alpha_t = \mathrm{normalize}(\hat{\alpha}_t)$. Note that $\sum_{j,n} \hat{\alpha}'_t(j,n)$ and $\sum_{j,n} \hat{\alpha}_t(j,n)$ are at most 1; this is a property of the forward algorithm (*cf.* [7, Section V.A]). Also, every element of $\hat{\alpha}'_t$, $\hat{\alpha}_t$ α'_t, and α_t is between 0 and 1. Thus, normalization moves each element of $\hat{\alpha}'_t$ and $\hat{\alpha}_t$ to the right on the number line, closer to 1, or leaves it unchanged. For concreteness, suppose $\sum_{j,n} \hat{\alpha}'_t(j,n) < \sum_{j,n} \hat{\alpha}_t(j,n)$; a completely symmetric argument applies when the inequality points the other way. This inequality implies that, on average, elements of $\hat{\alpha}'_t$ are to the left of elements of $\hat{\alpha}_t$. It also implies that, on average, normalization moves elements of $\hat{\alpha}'_t$ farther (to the right) than it moves elements of $\hat{\alpha}_t$. These observations together imply that, on average, corresponding elements of $\hat{\alpha}'_t$ and $\hat{\alpha}_t$ are closer to each other after normalization than before normalization, and hence that $||\alpha'_t - \alpha_t||_{\mathrm{sum}} \leq ||\hat{\alpha}'_t - \hat{\alpha}_t||_{\mathrm{sum}}$. Note that elements of $\hat{\alpha}'_t$ cannot move so much farther to the right than elements of $\hat{\alpha}_t$ that they end up being farther, on average, from the corresponding elements of $\hat{\alpha}_t$, because both matrices end up with the same average value for the elements (namely, $1/N_s N_m$).

Stricter Close-Enough Relation To improve the accuracy of the algorithm, a slightly stricter close-enough relation is used in our experiments: $\mathrm{closeEnough}(\alpha, \alpha')$ holds iff $||\alpha - \alpha'||_{\mathrm{sum}} \leq \epsilon \ \wedge \ (p_{\mathrm{dead}}(\alpha) = 0 \Leftrightarrow p_{\mathrm{dead}}(\alpha') = 0)$, where $p_{\mathrm{dead}}(\alpha)$ is the sum of the elements of α corresponding to compound states containing a dead state of M. It is easy to show that the algorithm still terminates, and that the above bound on cumulative inaccuracy still holds.

5 Predictive Analysis of Criticality Levels

Criticality Level. We define the *criticality level* of a monitor instance to be the inverse of the expected distance (number of steps) to a violation of the property of interest. To

compute this expected distance for each compound state, we compute a Discrete Time Markov Chain (DTMC) by composing the HMM model H of the monitored program with the DFSM M for the property. We then add a reward state structure to it, assigning a cost of 1 to each compound state. We use PRISM [6] to compute, as a reward-based reachability query, the expected number of steps for each compound state to reach compound states containing dead states of M. Note that these queries are issued in advance of the actual runtime monitoring, with the results stored in a table for efficient access.

Discrete-Time Markov Chain (DTMC). A Discrete-Time Markov Chain (DTMC) [6] is a tuple $\mathcal{D} = (S_D, \tilde{s}_0, \mathbf{P})$, where S_D is a finite set of states, $\tilde{s}_0 \in S_D$ is the initial state, and $\mathbf{P} : S_D \times S_D \to [0, 1]$ is the transition probability function. $\mathbf{P}(\tilde{s_1}, \tilde{s_2})$ is the probability of making a transition from $\tilde{s_1}$ to $\tilde{s_2}$.

Reward Structures. DTMCs can be extended with a reward (or cost) structure [6]. A *state reward function* $\underline{\rho}$ is a function from states of the DTMC to non-negative real numbers, specifying the reward (or cost, depending on the interpretation of the value in the application of interest) for each state; specifically, $\underline{\rho}(\tilde{s})$ is the reward acquired if the DTMC is in state $\tilde{s}$ for 1 time-step.

Composition of an HMM with a DFSM. Given an HMM $H = \langle S, A, V, B, \pi \rangle$ and a DFSM $M = \langle S_M, m_{init}, V, \delta, F \rangle$, their composition is a DTMC $\mathcal{D} = (S_D, \tilde{s}_0, \mathbf{P})$, where $S_D = (S \times S_M) \cup \{\tilde{s}_0\}$, $\tilde{s}_0$ is the initial state, and the transition probability function $\mathbf{P}$ is defined by:

- $\mathbf{P}(\tilde{s}_0, (s_i, m_{init})) = \pi$, with $1 \leq i \leq |S|$,
- $\mathbf{P}((s_{i_1}, m_{j_1}), (s_{i_2}, m_{j_2})) = A_{i_1,i_2} \sum_{\forall v_k \in V : \delta(m_{i_1}, v_k) = m_{i_2}} b_{i_1}(v_k)$.

We extend $\mathcal{D}$ with the state reward function such that $\underline{\rho}(\tilde{s}) = 1$ for all $\tilde{s} \in S_D$. With this reward function, we can calculate the expected number of steps until a particular state of the DTMC occurs.

Computing the Expected Distance. The expected distance $ExpDist(\tilde{s}, T)$ of a state $\tilde{s}$ of the DTMC to reach a set of states $T \subseteq S_D$ is defined as the expected cumulative reward and is computed as follows:

$$\mathrm{ExpDist}(\tilde{s}, T) = \begin{cases} \infty & \text{if } \mathrm{PReach}(\tilde{s}, T) < 1 \\ 0 & \text{if } \tilde{s} \in T \\ \underline{\rho}(\tilde{s}) + \sum_{\tilde{s}' \in S_D} \mathbf{P}(\tilde{s}, \tilde{s}') \cdot \mathrm{ExpDist}(\tilde{s}', T) & \text{otherwise} \end{cases}$$

where $\mathrm{PReach}(\tilde{s}, T)$ is the probability to eventually reach a state in T starting from $\tilde{s}$. For further details on quantitative reachability analysis for DTMCs, see [6]. The expected distance for a monitor instance with compound-state probability distribution α is then defined by $\mathrm{ExpDist}(\alpha, T) = \sum_{i,j} \alpha(i, j) \cdot \mathrm{ExpDist}((s_i, m_j), T)$.

6 Case Study

We evaluate our system by designing a monitor for the lock discipline property and applying it to the Btrfs file system. This property is implicitly parameterized by a `struct` type S that has a lock member, protected fields, and unprotected fields. Informally, the property requires that all accesses to protected fields occur while the lock is held.

The DFSM $M^{LD}(t, o)$ for the lock discipline property is parameterized by a thread t and an object o, where o is a particular `struct` with type S. There are four kinds of events: $\mathrm{LOCK}(t, o)$ (thread t acquires the lock associated with object o), $\mathrm{UNLOCK}(t, o)$ (thread t releases the lock associated with object o), $\mathrm{PROT}(t, o)$ (thread t accesses a protected field of object o), and $\mathrm{UNPROT}(t, o)$ (thread t accesses an unprotected field of object o). The DFSM $M^{LD}(t, o)$ is shown in the lower part of Figure 3(b); the parameters t and o are elided to avoid clutter. It requires that thread t's accesses to protected fields occur while thread t holds the lock associated with object o, except for accesses to protected fields before the first time t acquires that lock (such accesses are assumed to be part of initialization of o).

7 Implementation

Implementing the case study requires a gap-aware monitor and instrumentation that can intercept monitored events. Both these subsystems must integrate with our overhead control mechanism. The monitor must be able to recognize potential gaps caused by overhead control decisions, and the instrumentation must provide a means for the controller to disable monitoring by halting the interception of events. In addition, our implementation adapts to RVSE's criticality estimates by allocating hardware debugging resources to exhaustively monitor a small number of risky objects. This section discusses the implementation of these systems.

7.1 Gaps

On updating a monitor instance, the monitor processes a gap event before processing the current intercepted event if monitoring was disabled since the last time the monitor instance was updated. The gap event indicates that the monitor may have missed one or more events for the given instance during the time that monitoring was disabled.

The monitor determines whether a gap event is necessary by comparing the time of the last update to the monitor instance's state, which is stored along with the state, with the last time that monitoring was disabled for the current thread. For efficiency, we measure time using a counter incremented each time monitoring is disabled—a logical clock—rather than a real-time clock.

7.2 Instrumentation

For our case study, we monitor the lock discipline property for the `btrfs_space_info` struct in the Linux Btrfs file system. Each `btrfs_space_info` object has a spinlock, eight fields protected by the spinlock, and five fields not protected by the spinlock.

Using a custom GCC plug-in, we instrument every function that operates on a `btrfs_space_info` object, either by accessing one of its fields or by acquiring or releasing its spinlock. The instrumented function first has its function body *duplicated* so that there is an *active* path and an *inactive* path. Only the active path is instrumented for full monitoring. This allows monitoring to be efficiently enabled or disabled at the granularity of a function execution. Selecting the inactive path effectively disables monitoring. When a duplicated function executes, it first calls a *distributor* function that calls the overhead control system to decide which path to take. We enable and disable monitoring at the granularity of function executions, because deciding to enable or disable monitoring at the granularity of individual events would incur too much overhead.

Every `btrfs_space_info` operation in the active path is instrumented to call the monitor, which updates the appropriate monitor instance, based on the thread and the `btrfs_space_info` object involved. For fast lookup, all monitor instances associated with a thread are stored in a hash table local to that thread and indexed by object address.

7.3 Hardware Supervision

Our system prioritizes monitoring of objects with high criticality by placing them under hardware supervision. Specifically, we use debug registers to monitor every operation on these objects even when other monitoring is disabled (i.e., when the inactive path is taken). The debug registers cause the CPU to raise a debug exception whenever an object under hardware supervision is accessed, allowing the monitor to observe the access. Note that this allows monitoring to be enabled and disabled on a per-object basis, for a limited number of objects, in contrast to the per-function-execution basis described above. The overhead remaining after monitoring the hardware supervised objects is distributed to the other objects in the system using the normal overhead control policy.

Our current implementation keeps track of the most critical object in each thread. Each thread can have its own debug register values, making it possible to exhaustively track events for one monitor instance in each thread for any number of threads.

Because an x86 debug register can at most watch one 64-bit memory location, we need a small amount of additional instrumentation to monitor all 13 fields in a supervised `btrfs_space_info` object. Our plug-in instruments every `btrfs_space_info` field access in the *inactive* path with an additional read to a dummy field in the same object. Setting the debug register to watch the dummy field of a supervised object causes the program to raise a debug exception whenever any field of that object is accessed from the inactive path. The debug exception handler calls the monitor to update the monitor instance for the supervised object.

For `btrfs_space_info` spinlock acquire and release operations, we instrument the inactive path with a simple check to determine if the spinlock belongs to one of the few supervised objects that should be updated even though monitoring is disabled. We could use debug registers to remove the need for this check, but we found that overhead from checking directly was very low, because lock operations occur infrequently compared to field accesses.

7.4 Training

We collected data from completely monitored runs to train the HMM and learn the gap length distribution. During training runs for a given overhead level, the distributor makes monitoring decisions as if overhead control were in effect but does not enforce those decisions; instead, it always takes the active path. As a result, the system knows which events would have been missed by taking the inactive path. Based on this information, for each event that would have triggered processing of a gap event, we compute the actual number of events missed for the corresponding monitor instance. The gap length distribution for the given overhead level is the distribution of those numbers.

Our case study uses a simple overhead-control mechanism in which the target "overhead level" is specified by the fraction f of function executions to be monitored. For each function execution, the distributor flips a biased coin, which says "yes" with probability f, to decide whether to monitor the current function execution. We tested three different sampling probabilities: 50%, 75%, 85%, and 95%. For each sampling probability, we precomputed the RVSE distributions with $\epsilon = 0.1$, thereby obtaining four RVSE graphs having 12,177, 33,234, 30,645 and 11,622 nodes, respectively.

7.5 Evaluation

We used two different tests to measure how well our prioritization mechanism improved ARV's effectiveness. The first test runs with an unmodified version of Btrfs, which does not contain any lock discipline violations, in order to test how well prioritization avoids false alarms. The second test runs on a version of Btrfs with an erroneous access that we inserted, to test if prioritization improves our chances of detecting it. For both of these tests, we run Racer [12], a workload designed specifically to stress file system concurrency, on top of a Btrfs-formatted file system, and we report results that are averaged over multiple runs.

We tested three configurations: 1) hardware supervision disabled, 2) randomly assigned hardware supervision, and 3) adaptive hardware supervision that prioritizes critical objects, as described above. Most threads in the Racer workload had two associated monitor instances. At any time, our prioritization chose one of those from each thread to supervise.

The table below shows the results for these tests. Each row in the table is for one of the three sampling probabilities. For our false alarm test, the columns labeled FalseAlarm in the table show how many monitor instances had an error probability higher than 0.8 at the end of the run. Because the run had no errors, lower numbers are better in this test. For our error detection test, we checked the corresponding monitor instance immediately after our synthetic error triggered; the columns labeled ErrDet in the table show the percentage of the times that we found that monitor instance to have an error probability higher than 0.8, indicating it correctly inferred a likely error. For this test, higher numbers are better. All results are averaged over multiple runs.

In all cases, hardware supervision improved the false alarm rate and the error detection rate. For the 75% and 85% sampling profiles, adaptive prioritization provides greater improvement than simply choosing objects at random for supervision. With 50% sampling, adaptive sampling does worse than random, however. In future work,

Sampling	No Supervision		Random Supervision		Adaptive Supervision	
Probability	FalseAlarm	ErrDet	FalseAlarm	ErrDet	FalseAlarm	ErrDet
50%	30.3	23.0%	11.7	57.4%	12	50.1%
75%	47	31.2%	36	69.3%	17	79.4%
85%	5502	34.1%	5606	72.3%	5449	85.1%

we intend to improve our criticality metric so that it performs better at lower overheads. The table also shows that ARV takes advantage of increased sampling rates, successfully detecting more errors in the error detection test. We are currently investigating why performance in the false alarm test declines with higher sampling rates.

8 Related Work

In [2], the authors propose a method for the automatic synthesis and adaptation of invariants from the observed behavior of an application. Their overall goal is adaptive application monitoring, with a focus on interacting software components. In contrast to our approach, where we learn HMMs, the invariants learned are captured as finite automata (FA). These FA are necessarily much larger than their corresponding HMMs. Moreover, error uncertainty, due to inherently limited training during learning, must be dealt with at runtime, by modifying the FA as needed. They also do not address the problem of using the synthesized FA for adaptive-control purposes.

A main aspect of our work is our approximation of the RVSE forward algorithm for state estimation, which pre-computes compound-state probability distributions and stores them in a graph. In the context of the runtime monitoring of HMMs, the authors of [10] propose a complementary method for accelerating the estimation of the current (hidden) state: Particle filters [4]. This sequential Monte-Carlo estimation method is particularly useful when the number of states of the HMM is very large, in particular, much larger than the number of particles (i.e., samples) necessary for obtaining a sufficiently accurate approximation. This, however, is typically not the case in our setting, where the HMMs are relatively small. Consequently, the Particle filtering method would have introduced at least as much overhead as the forward algorithm, and would have therefore also required a priori (and therefore approximate) state estimation.

The runtime verification of HMMs is explored in [9, 3], where highly accurate deterministic and randomized methods are presented. In contrast, we are considering the runtime verification of actual programs, while using probabilistic models of program behavior in the form of HMMs to fill in gaps in execution sequences.

9 Conclusions

We have presented Adaptive Runtime Verification, a new approach approach to runtime verification that synergistically combines overhead control, runtime verification with state estimation, and predictive analysis of monitor criticality levels. We have demonstrated the utility of the ARV framework through a significant case study involving the monitoring of concurrency errors in the Linux kernel.

Future work will involve extending the ARV framework with a recovery mechanism that will come into play when a property violation is detected or imminent. We will also consider additional case studies, including those that use SMCO [5] for their overhead control. Fully integrating SMCO will require a new method to compute the probability distribution on the length of gaps introduced by SMCO for any given target overhead.

Acknowledgements. We thank the anonymous reviewers for their valuable comments. Research supported in part by AFOSR Grant FA9550-09-1-0481, NSF Grants CCF-1018459, CCF-0926190, and CNS-0831298, and ONR Grant N00014-07-1-0928.

References

1. Baum, L.E., Petrie, T., Soules, G., Weiss, N.: A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. The Annals of Mathematical Statistics 41(1), 164–171 (1970)
2. Denaro, G., Mariani, L., Pezze, M., Tosi, D.: Adaptive runtime verification for autonomic communication infrastructures. In: Proc. of the International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM). vol. 2, pp. 553–557. IEEE Computer Society (2005)
3. Gondi, K., Patel, Y., Sistla, A.P.: Monitoring the full range of omega-regular properties of stochastic systems. In: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'09). pp. 105–119. Savannah, GA, USA (2009)
4. Gordon, N., Salmond, D., Smith, A.: Novel approach to nonlinear/non-Gaussian Bayesian state estimation. In: IEEE Proceedings on Radar and Signal Processing. vol. 140, pp. 107–127. IEEE (1993)
5. 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 (STTT) 14(3), 327–347 (2012)
6. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Formal Methods for Performance Evaluation. Lecture Notes in Computer Science, vol. 4486, pp. 220–270. Springer (2007)
7. Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE 77(2), 257–286 (1989)
8. Seyster, J., Dixit, K., Huang, X., Grosu, R., Havelund, K., Smolka, S.A., Stoller, S.D., Zadok, E.: InterAspect: Aspect-oriented instrumentation with GCC. Formal Methods in System Design (2012), to appear
9. Sistla, A.P., Srinivas, A.R.: Monitoring temporal properties of stochastic systems. In: Proceedings of the Ninth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'08). pp. 185–212. San Francisco, CA, USA (2008)
10. Sistla, A., Zefran, M., Feng, Y.: Monitorability of stochastic dynamical systems. In: Proc. 23rd International Conference on Computer Aided Verification (CAV 2011). Lecture Notes in Computer Science, vol. 6806, pp. 720–736. Springer (2011)
11. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: Proc. 2nd International Conference on Runtime Verification (RV'11). San Fransisco, CA (September 2011), (**Won best paper award**)
12. Subrata Modak: Linux Test Project (LTP) (2009), `http://ltp.sourceforge.net/`