Monitoring with uncertainty

Ezio Bartocci
Faculty of Informatics
Vienna University of Technology
Vienna, Austria
ezio.bartocci@tuwien.ac.at

Radu Grosu
Faculty of Informatics
Vienna University of Technology
Vienna, Austria
radu.grosu@tuwien.ac.at

We discuss the problem of runtime verification of an instrumented program that misses to emit and to monitor some events. These gaps can occur when a monitoring overhead control mechanism is introduced to disable the monitor of an application with real-time constraints. We show how to use statistical models to learn the application behavior and to “fill in” the introduced gaps. Finally, we present and discuss some techniques developed in the last three years to estimate the probability that a property of interest is violated in the presence of an incomplete trace.

1 Problem description

Runtime verification (RV) (or *monitoring*) [9, 2] is a well-established technique to check whether the current execution of a program satisfies a property of interest. In the last years, RV has increasingly gained popularity in both the formal verification and software engineering communities, because in most of the cases it reveals to be more practical than exhaustive verification techniques such as model checking [5, 12] and more versatile than classical software testing. Formally, the RV problem is to decide whether an execution trace τ of a program P satisfies a temporal logic specification φ. RV is then usually performed by translating the formula φ into a deterministic finite state machine (DFSM) M_φ and by instrumenting the program P so that it emits the events triggering the input-enabled transitions in M_φ.

However, RV does not come for free. It introduces runtime overhead, thereby changing the timing-related behavior of the program under scrutiny. While this is acceptable in many applications, it may be unacceptable in applications with real-time constraints. In such cases, overhead control is necessary. Recently, a number of techniques have been developed to mitigate the overhead due to RV [4, 6, 8, 1, 10]. 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. In a previous work [10], 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.

For example, let φ be the formula $\Box(a \Rightarrow \Diamond c)$, that means always an event a is finally followed by an event c and let τ be the trace $a\ b\ b\ c\ a\ d\ b\ c$. In this example the formula φ clearly holds. Suppose now that the trace is *incomplete*, due to disabled monitoring: $\tau = a\ b\ b\ c - b\ c$, with $-$ indicating a set of events that could not be observed (or gap). Although it is not possible to be really sure that the property is satisfied, the problem we are interested to solve is to quantify the uncertainty with which the trace τ satisfies φ.

2 “Filling in” the gaps

In order to quantify the uncertainty in monitoring, 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 of interest is satisfied or violated. In our previous works [15, 3, 11], we have chosen the Dynamic Bayesian Networks (DBNs) [14], a suitable formalism to characterize the temporal probability model of the instrumented program emitting events. DBNs can have multiple state variables (modeling the states of a program that are usually hidden) and multiple observation variables (representing the output events). A DBN is essentially a first-order Markov Process where each variable at time t can depend only by other variables at the same time t or at time $t-1$.

In [15, 3] we used Hidden Markov Models (HMM), that are essentially DBNs with only one state variable and one observation variable. A 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 a 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 generates a given observation sequence.

One can obtain a HMM for a system automatically, by learning it from complete traces using standard HMM learning algorithms(i.e. Baum–Welch) [13]. These algorithms require the user to specify the desired number of states in the HMM and they 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 close to zero. This information can help the learning algorithm converge more quickly and find globally (instead of locally) optimal solutions.

In contrast to our previous work [15, 3], in [11] we succinctly represent the program model, the program monitor, their interaction, and their observations as a generic DBN. This allowed 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 DFSM. We use peek events at the end of monitoring gaps to refocus the DBN and DFSM states. Our combination of these two kind of observations, program events and peek events, is akin to *sensor fusion* in robotics.

3 Monitoring with uncertainty

Runtime Verification with State Estimation (RVSE)

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) [15], in which a HMM is used to succinctly model the program and 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 state machine (DFSM). 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 DFSM. 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.

Approximate Precomputed Runtime Verification with State Estimation (AP-RVSE)

To reduce the runtime overhead, we developed a version of the algorithm, which we call the approximate precomputed RVSE (AP-RVSE), which pre-computes the matrix calculations and stores the results in a table [3]. Essentially, AP-RVSE pre-computes 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 ε parameter: if a newly computed matrix differs from the matrix on an existing node by at most ε according to the 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.

Runtime Verification with Particle Filtering (RVPF)

In a recent paper [11] we have introduced an alternative approach, called Runtime Verification with Particle Filtering (RVPF), to control the balance between runtime overhead, memory consumption, and prediction accuracy. In one of the most common forms of 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 to the states to provide a better prediction of the program events. We exploit the knowledge of the current program event and the particular structure of the DBN to improve the variance of the PF, by using sequential importance resampling (SIR). In this PF variation, resampling (which is a major 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.

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 in [11] along with our previous two algorithms in C and compared them through experiments based on the benchmarks used in [3]. Our results confirm RVPF's versatility.

Acknowledgements. The main concepts and ideas presented in this invited talk were developed (and presented in several papers) in collaboration with Scott A. Smolka, Scott D. Stoller, Erez Zadok, Justin Seyster and Klaus Havelund that we would like here to acknowledge.

References

[1] Matthew Arnold, Martin Vechev & Eran Yahav (2008): *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)*, ACM, pp. 143–162, doi:10.1145/1449955.1449776.

[2] H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G.J. Pace, G. Rosu, O. Sokolsky & N. Tillmann (2010): *Preface*. In: *Proc. of RV 2010, the First International Conference on Runtime Verification, St. Julians, Malta, November 1-4, 2010*, *Lecture Notes in Computer Science* 6418, Springer, doi:10.1007/978-3-642-16612-9.

[3] E. Bartocci, R. Grosu, A. Karmarkar, S. A. Smolka, S. D. Stoller, E. Zadok & J. Seyster (2012): *Adaptive Runtime Verification*. In: *Proc. of RV 2012, the third International Conference on Runtime Verification, September, 2012 Istanbul, Turkey*, *Lecture Notes in Computer Science* 7687, Springer, pp. 168–182, doi:10.1007/978-3-642-35632-2_18.

[4] B. Bonakdarpour, S. Navabpour & S. Fischmeister (2011): *Sampling-Based Runtime Verification*. In: *Proc. FM 2011: Formal Methods, the 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011*, *Lecture Notes in Computer Science* 6664, Springer, pp. 88–102, doi:10.1007/978-3-642-21437-0_9.

[5] E. M. Clarke & E. Emerson (1982): *Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic*. In Dexter Kozen, editor: *Logics of Programs*, *Lecture Notes in Computer Science* 131, Springer Berlin / Heidelberg, pp. 52–71, doi:10.1007/BFb0025774.

[6] L. Fei & S.P. Midkiff (2006): *Artemis: Practical Runtime Monitoring of Applications for Execution Anomalies*. In: *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)*, ACM, Ottawa, Canada, pp. 84–95, doi:10.1145/1133981.1133992.

[7] N.J. Gordon, D.J. Salmond & A.F.M. Smith (1993): *Novel approach to nonlinear/non-Gaussian Bayesian state estimation*. In: *IEEE Proceedings on Radar and Signal Processing*, 140, IEEE, pp. 107–127.

[8] M. Hauswirth & T. M. Chilimbi (2004): *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, doi:10.1145/1024393.1024412.

[9] K. Havelund & G. Rosu (2002): *Runtime Verification, RV 2002: Preface*. *Electr. Notes Theor. Comput. Sci.* 70(4), pp. 201–202, doi:10.1016/S1571-0661(05)80585-7.

[10] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller & E. Zadok (2012): *Software Monitoring with Controllable Overhead*. *STTT* 14(3), pp. 327–347, doi:10.1007/s10009-010-0184-4.

[11] K. Kalajdzic, E. Bartocci, S. A. Smolka, Scott Stoller & G. Grosu (2013): *Runtime Verification with Particle Filtering*. In: *Proc. of RV 2013, the fourth International Conference on Runtime Verification, INRIA Rennes, France, 24-27 September, 2013*, Lecture Notes in Computer Science, Springer, p. To Appear.

[12] J.P. Queille & J. Sifakis (1982): *Specification and verification of concurrent systems in CESAR*. In: *Proc. of the 5th Colloquium on International Symposium on Programming*, Springer-Verlag, pp. 337–351, doi:10.1007/3-540-11494-7_22.

[13] Lawrence R. Rabiner (1989): *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*. *Proceedings of the IEEE* 77(2), pp. 257–286, doi:10.1109/5.18626.

[14] Stuart Russell & Peter Norvig (2010): *Artificial Intelligence: A Modern Approach*, 3rd edition. Prentice-Hall.

[15] S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka & E. Zadok (2011): *Runtime Verification with State Estimation*. In: *Proc. of RV 2011, the Second international conference on Runtime verification, San Francisco, CA, USA*, *Lecture Notes in Computer Science* 7186, Springer-Verlag, pp. 193–207, doi:10.1007/978-3-642-29860-8_15.