Proceedings of the
2nd Workshop on Artificial Intelligence and Formal Verification, Logics, Automata and Synthesis (OVERLAY),
September 25, 2020

Learning How to Monitor: Pairing Monitoring and Learning for Online System Verification *

Andrea Brunello[1], Dario Della Monica[1], Angelo Montanari[1], and Andrea Urgolo[2]

[1,2]University of Udine, Italy
[1]{andrea.brunello,dario.dellamonica,angelo.montanari}@uniud.it
[2]urgolo.andrea@spes.uniud.it

Abstract

In several domains, the execution of a system is associated with the generation of continuous streams of data. Such streams may contain important telemetry information, which can be used to perform tasks like predictive maintenance and preemptive failure detection, in order to issue early warnings. In critical contexts, formal methods have been recognized as an effective approach to ensure the correct behaviour of a system. However, they have at least two significant weaknesses: (*i*) a complete, hand-made specification of all the properties that have to be guaranteed during the execution of the system turns out to be often out of reach when complex systems have to be handled and, for the same complexity reasons, (*ii*) it may be difficult to derive a complete model of the system against which to check the properties of interest. In this paper, to overcome these limitations, we extend a recently presented framework that pairs monitoring with machine learning, in order to allow for the preemptive detection of critical system behaviours in an on-line setting. The framework is tested on a practical use-case based on the public NASA C-MAPSS dataset, and is shown to obtain promising performance in terms of its ability to forecast the approach of failures, and to provide interpretable results.

1 Introduction

Typically, during its execution a system generates several streams of data, which may contain important telemetry information. This is the case, for instance, with logs produced by web servers, smart sensors, or machinery in modern industrial plants. System behaviours may be arbitrarily convoluted, as they can be the result of the interaction among several components as well as with the surrounding environment.

In such a complex setting, formal methods can be exploited as effective tools for the automatic verification of software and hardware systems, a task which is of paramount importance in many critical domains. However, the inherent complexity of system's components and of their interactions make it very difficult (and sometimes impossible) to specify in advance all the relevant properties that have to be guaranteed (or, dually, avoided) during their execution. In addition, the definition of a complete model of the system against which to check the properties of interest may also be out of reach.

*The authors acknowledge the partial support by the Italian INdAM-GNCS project *Ragionamento Strategico e Sintesi Automatica di Sistemi Multi-Agente*.

To overcome these limitations, some approaches that complement formal verification with model-based testing and monitoring have been recently proposed in the literature (see, for instance, [5, 6]). In this work, we focus on *monitoring* [8], a runtime verification technique whose key feature is that of allowing one to detect the satisfaction or violation of a property (usually expressed in terms of some temporal logic formula) by analyzing a single run of the system, which makes such a technique naturally applicable to data streaming contexts. Specifically, we extend a recently-proposed framework for online system verification [3] that integrates monitoring with machine learning and can be applied to preemptive failure detection and predictive maintenance tasks in data streaming contexts. The framework is evaluated on a use-case based on the public NASA C-MAPSS (Commercial Modular Aero-Propulsion System Simulation) dataset [14], and is shown to obtain promising results regarding the preemptive detection of engine failures. Moreover, the approach is highly interpretable, meaning that a domain expert can easily read and validate the generated model. Last but not least, by looking at the formulas extracted by the framework, it may be possible to gain some insights on the causes that ultimately led to a failure, and act accordingly.

2 Learning How to Monitor

As we already pointed out, the framework combines machine learning with monitoring, in order to obtain a system that can be exploited for online system verification. Its operation consists of five main steps.

1. **Specification of the initial set of properties.** Domain experts are asked to specify the most significant (monitorable) properties that the system under consideration should exhibit. The latter are then formalized in a suitable temporal logic and a monitor that checks them against incoming execution traces is synthesized.

2. **Monitoring of system properties.** The monitor checks whether the system satisfies/violates the specified properties during its execution.

3. **Detection of a failure.** Upon the detection of a failure, the part of the system trace for which the monitor reaches a verdict of failure is extracted, and considered to be a *failure trace*. Intuitively, it corresponds to the subtrace that is closer to the failure event. In addition, the remaining part of the trace, generated by the system during previous normal execution is extracted, and considered to be a *normal* trace. Of course, the length of the time window that is used to distinguish between normal and failure traces depends on the specific application domain, and it must be carefully chosen according to the results of a dedicated tuning phase, possibly with the help of domain experts.

4. **Mining of the relevant behaviour patterns.** Failure and normal traces are put together to generate a dataset for supervised machine learning. Each of the two instances is characterized by a (possibly multivariate) trace that can contain numerical (as in the case of a temperature signal) or categorical (e.g., a sequence of system calls made in a Unix system) values. Traces are first converted into timelines (see, e.g., [9, 15]); then, pattern mining algorithms are run, with the goal of extracting the (temporal) logic formulas that best characterize and discriminate between normal and failure traces (following, for instance, the approaches described in [2, 4, 7, 10]). In doing that, some attention has to be paid to avoid overfitting, for instance putting a constraint on the maximum nesting level of the extracted formulas, or considering only a set of predefined patterns (see, e.g., [12]).

5. **Extension of the pool of properties.** The temporal logic formulas extracted during the mining phase are added to the pool of properties to be monitored, and the process restarts from the monitoring phase. Notice that it might be possible for the added formulas to be redundant (i.e., entailed by other ones) but they cannot be in contradiction with the existing ones.

The framework works in an iterative way, which we may refer to as its *online* phase, in which incoming traces are considered. It starts from a set of basic properties, and new, related ones are then added over time, with the idea that, in principle, they should allow the system to discover anomalous behaviours earlier and with ever increasing accuracy and coverage. In order for the pool of properties to converge,

we are investigating the possibility to define some stopping criteria, e.g., based on the accuracy of the extracted formula or the complexity of the corresponding decision trees. Based on the organization of the above five steps, two different framework phases and learning modes may be identified.

Warmup and online execution phases. Sometimes, data pertaining to past system failures may be available, or approximate data may be generated by means of simulations. In that case, it makes sense to exploit these pieces of information to perform monitor learning even before its *online* phase. To do that, intuitively, it is sufficient to mimic the continual arrival of the available traces, and to iteratively follow the five steps which we described. Thanks to this initial *warmup* phase, the framework can then deal with the subsequent *online* phase starting with an already-extended pool of properties.

Semi-supervised and unsupervised learning. Let us focus on the task of preemptive machinery fault detection. According to the first step of the framework, a domain expert may be required to specify an initial set of properties to be monitored against the execution of the system, thus acting, in her/his vision, as fault early warnings. Since there is, at first, a human intervention, we can refer to this strategy as a kind of *semi-supervised* learning. Nevertheless, domain expert knowledge may sometimes not be available. In such a case, the monitor is initialized with just a single, trivial property, that is, "the machinery is in operation". Then, once a failure is detected (indeed not in a preemptive way), the framework may proceed with the usual steps in order to detect some properties that may help it in forecasting the fault before it actually happens. Since in this operation mode there is no human intervention (except for the trivial, initial, "machinery in operation" property), we can refer to it as a kind of *unsupervised* learning.

3 Application

In order to better describe the operation of the framework, we now turn our attention to its application to the NASA C-MAPSS FD001 dataset [14], which includes run-to-failure simulated data of turbo fan jet engines and is considered as a benchmark in the literature (see, for example, [13]). Specifically, in dataset FD001, engines are simulated according to a single condition (called *Sea level*), and their failures are attributable to one possible cause (HPC degradation). Each engine simulation is represented by a multivariate time series obtained from 21 engine's sensors and 3 operational settings. Although each instance represents the simulation of a different engine, the data can be considered to be from a fleet of engines of the same type. Data are sampled at one value per second, and the average time series length is 206. The dataset includes 100 training and 100 test instances. However, test instances are not ideal for our purposes as they do not end at a failure, but at an arbitrary preceding point. Thus, for the sake of this early stage evaluation, we focus on training set instances with at least 200 points, and we randomly split them into training and test instances. As a result, we end up using 30 training and 18 test samples.

The main goal of this experiment is the extraction of temporal properties in order to enhance the preemptiveness of engine failure predictions. In order to encode the temporal properties extracted by the framework, we chose here to rely on Linear Temporal Logic (LTL) [11] and, for this reason, before applying the framework, time series are converted into discrete-valued timelines. Intuitively, given a real-valued time series, we want to translate it into a sequence of symbols belonging to a finite dictionary. In order to perform the translation, we chose to rely on Symbolic Aggregate approXimation (SAX) [9], a well-established solution which has already been used in several data mining applications.

3.1 Experiment setup

As for the execution of the framework, we proceed as follows. The monitoring pool is set up with just a single, trivial, "engine in operation" property (*unsupervised* setting). Then, the arrival of engine telemetry data is simulated, as in the *warmup* phase. To do so, considering that only one kind of engine is included in the dataset, we act as if we were observing a single instance: we concatenate all training timelines one after the other, choosing their order by means of a random seed. Then, the monitoring phase is started, and the framework is fed with the incoming data one point at a time. Upon the detection of a failure, two sub-traces are extracted: the *failure window*, which is the portion of the time series that immediately precedes the failure event, and the *normal* trace, which lasts from the beginning of the timeline or from the last failure occurrence until the *failure window*. Next, contrastive LTL specifications describing how

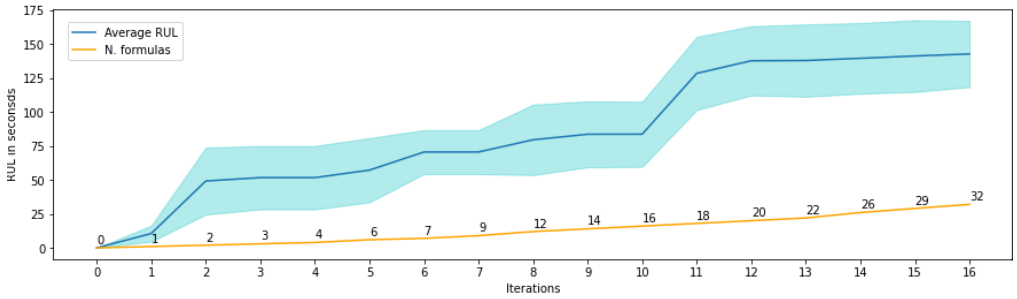

Figure 1: Average and standard deviation of RUL, and number of LTL formulas in the monitoring pool for each framework iteration (*warmup* phase).

the two traces differ are generated by means of a publicly available tool [1] based on a probabilistic Bayesian generative model [7] which exploits a Markov Chain Monte Carlo (MCMC) algorithm with linear complexity with reference to traces length $\times$ number of iterations. Such specifications are based on a set of predefined LTL templates representing the basic operators and properties *globally, eventually, until, release, response, stability*, and *sometimes before*. It should be noticed that the generated formulas capture high-level qualitative temporal relations and are unable to fully store the analyzed traces (e.g., formulas like $p_1 \wedge Xp_2 \wedge XXp_3 \wedge XXXp_4 \ldots$ are never generated), thus they are not able to distinguish between *failure* and *normal* sub-traces based on their difference in length. At this point, by means of a decision tree trained to distinguish between the two kinds of traces, the most useful specifications are selected and combined into two Boolean formulas. Finally, once fitted, the resulting decision tree is added to the monitoring pool. After that, the monitoring process resumes its operation on the remaining concatenated data, as if the engine was fixed and started again. Each time the monitor reaches a verdict of failure (due to a decision tree in the pool predicting it, or to observing an actual engine breakdown), the remaining useful lifetime (RUL) of the engine is calculated. The entire process is run 100 times varying the random seed, so to collect statistical data regarding the evolution of RUL. In this way, based on the collected statistics, it is possible to decide when to interrupt the warmup phase of the framework, according to the desired preemptiveness level of failure prediction. Operationally, for the procedure we empirically chose a time window of 30 seconds and a desired preemptiveness threshold of 140 seconds.

3.2 Results

As shown in Figure 1, in the warmup phase the 140 second threshold is reached after 16 iterations of the framework, with a standard deviation of 24.34 seconds and 32 LTL formulas learned. Moreover, it can be noticed that a law of diminishing returns applies regarding the pool size and RUL estimates. In order to confirm the consistency of the RUL values, we applied one of the trained monitors to the 18 test instances, obtaining an average RUL of 142.0 seconds with a standard deviation of 25.36. Considering the interpretability of the results, we may refer to the formulas $f_1 = (sensor_{12}\_B \wedge \neg sensor_2\_E)\mathcal{R}\ \neg sensor_2\_E$ and $f_2 = sensor_{15}\_D\ \mathcal{U}\ sensor_2\_E$ learned respectively in the iteration 1 and 2: f_1 means that if $sensor_2\_E$ occurs then $sensor\_12\_B$ occurred in the past and f_2 means that $sensor_{15}\_D$ has to be true until $sensor_2\_E$ eventually becomes true. A monitor execution on a test set trace is shown in Figure 2. While f_1 identifies the window immediately preceding the failure, f_2, which

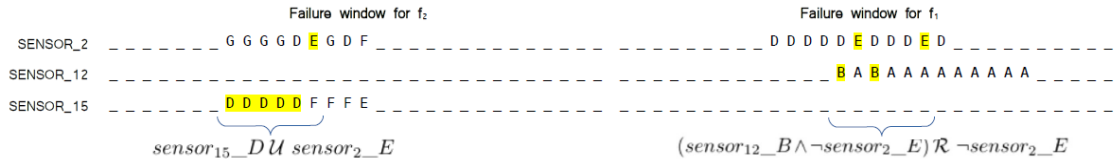

Figure 2: Example of a trace portion with predicted failure windows.

was extracted at the second iteration, allows us to anticipate the prediction to an earlier instant.

4 Conclusions and future work

In this paper, a novel framework that integrates machine learning with monitoring to perform tasks such as predictive maintenance and preemptive failure detection in an online setting is discussed. One of its major strengths and distinguishing characteristics with respect to other solutions is the interpretability of the extracted properties that are used to predict the failures. The framework, tested on the NASA C-MAPSS FD001 dataset, is shown to provide encouraging results. As for future work, it includes (*i*) the development of a metric that would allow to estimate the RUL of the monitored system without relying on the warmup phase, (*ii*) the testing of other, more sophisticated time series preprocessing techniques, and (*iii*) a thorough comparison with other state-of-the-art predictive maintenance approaches.

References

[1] BayesLTL GitHub reference page. https://github.com/IBM/BayesLTL. Accessed online on 21 August 2020.

[2] D. Bresolin, E. Cominato, S. Gnani, E. Muñoz-Velasco, and G. Sciavicco. Extracting interval temporal logic rules: A first approach. In *Proceedings of the 25th International Symposium on Temporal Representation and Reasoning*, volume 120 of *LIPIcs*, pages 7:1–7:15, 2018.

[3] A. Brunello, D. Della Monica, and A. Montanari. Pairing monitoring with machine learning for smart system verification and predictive maintenance. In *Proceedings of the 1st Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis*, volume 2509 of *CEUR Workshop Proceedings*, pages 71–76, 2019.

[4] A. Brunello, G. Sciavicco, and I. E. Stan. Interval temporal logic decision tree learning. In *Proceedings of the 16th European Conference on Logics in Artificial Intelligence*, volume 11468 of *Lecture Notes in Computer Science*, pages 778–793, 2019.

[5] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. In *Proceedings of the 2nd International Workshop on Pre- Post-Deployment Verification Techniques*, pages 15–28, 2017.

[6] M. Gerhold, A. Hartmanns, and M. Stoelinga. Model-based testing of stochastically timed systems. *Innovations in Systems and Software Engineering*, 15(3-4):207–233, 2019.

[7] J. Kim, C. Muise, A. Shah, S. Agarwal, and J. Shah. Bayesian inference of linear temporal logic specifications for contrastive explanations. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 5591–5598. AAAI Press, 2019.

[8] M. Leucker and C. Schallhart. A brief account of Runtime Verification. *Journal of Logical and Algebraic Methods in Programming*, 78(5):293–303, 2009.

[9] J. Lin, E. Keogh, L. Wei, and S. Lonardi. Experiencing SAX: A novel symbolic representation of time series. *Data Mining and knowledge discovery*, 15(2):107–144, 2007.

[10] D. Neider and I. Gavran. Learning linear temporal properties. In *Proceedings of the 18th Conference on Formal Methods in Computer Aided Design*, pages 1–10, 2018.

[11] A. Pnueli. The temporal logic of programs. In *Proceedins of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[12] Property pattern mappings for LTL. https://matthewbdwyer.github.io/psp/. Kansas State University CIS Department, Laboratory for Specification, Analysis, and Transformation of Software (SAnToS Laboratory) – accessed online on 17 July 2020.

[13] E. Ramasso and A. Saxena. Performance benchmarking and analysis of prognostic methods for CMAPSS datasets. *International Journal of Prognostics and Health Management*, 5:1–15, 11 2014.

[14] A. Saxena, K. Goebel, D. Simon, and N. Eklund. Damage propagation modeling for aircraft engine run-to-failure simulation. In *Proceedings of the 2008 International Conference on Prognostics and Health Management*, pages 1–9. IEEE, 2008.

[15] G. Sciavicco, I. E. Stan, and A. Vaccari. Towards a general method for logical rule extraction from time series. In *Proceedings of the 8th International Work-Conference on the Interplay Between Natural and Artificial Computation*, volume 11487 of *Lecture Notes in Computer Science*, pages 3–12, 2019.