Pairing monitoring and machine learning for early failure detection and predictive maintenance

Angelo Montanari
(with A. Brunello, L. Geatti, and N. Saccomanno)

IRIS-AI

Formal methods at work

Formal methods make it possible to check the consistency of specifications (**satisfiability checking**) and to ensure the correct behaviour of a (model of a) system against a specification (**model checking**) in an effective way.

In many critical contexts, they are **highly desirable**.

However, specifying in advance all the relevant properties and building a complete model of the system against which to check them is often **out of reach** in real-world scenarios.

Pairing monitoring and machine learning

To overcome these limitations, we developed a framework that

- supports **monitoring**, a lightweight runtime verification technique that does not require an explicit model of the system under consideration, and
- pairs it with a **machine learning tool** to automatically extend the set of properties to monitor on the basis of historical trace data. Many systems have, indeed, such a level of complexity that it is impossible for a system engineer to specify in advance all properties to be monitored.

Observation. Monitoring does not replace the above-mentioned formal methods; rather, it integrates satisfiability checking and model checking.

Early failure detection and predictive maintenance

A **failure** can be viewed as a deviation between the observed behavior and the required behavior of the system.

Early failure detection refers to the process of identifying potential issues or faults in a system before they escalate into major problems or failures.

Such a proactive approach is crucial not only to prevent catastrophic failures, but also to minimize downtime, thus reducing maintenance costs (**predictive maintenance**).

Outline of the presentation

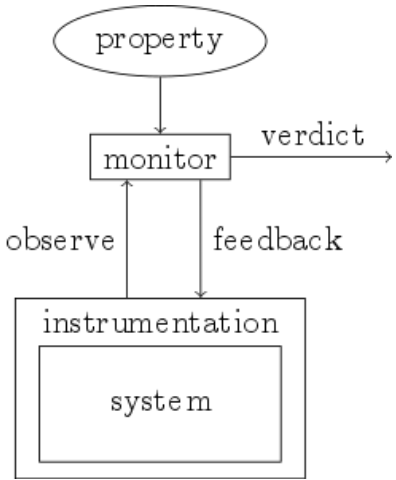
In the following, I give a short account of

- monitoring and monitorability
- integration of monitoring and machine learning
- current and future research and development directions

Monitoring is a runtime verification technique for the formal analysis of systems that checks a **finite** prefix of the current execution (*trace*) of the *system under scrutiny* to detect failures or successes expressed by means of temporal formulas.

The verdict of a monitoring algorithm is **irrevocable**: once a failure (resp., a success) is detected, **all** continuations of the execution of the system are guaranteed to be failures (resp., successes).

A simple architecture



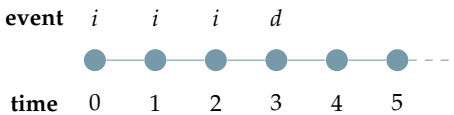
Monitoring typically consists of the following steps:

1. bad/good behaviors to be checked are specified by temporal logic formulas;
2. from each temporal formula, an equivalent *monitor* is built (typically, a deterministic finite state automaton);
3. the monitor is used for analysing the system in either online or offline mode.

Positively and negatively monitorable properties

Negatively monitorable

Property: *"The program never enters a deadlock"*

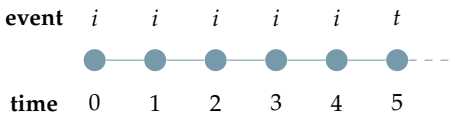


i = instruction
d = deadlock

Given a *safety* property, a prefix of a sequence suffices to establish whether it *does not* satisfy the property.

Positively monitorable

Property: *"The program terminates"*



i = instruction
t = termination

Given a *cosafety* property, a prefix of a sequence suffices to establish whether it *does* satisfy the property.

Linear Temporal Logic (LTL)

Let $\mathcal{AP} = \{p, q, r, \dots\}$ be a set of *atomic propositions*. The syntax of LTL is defined as follows:

$$\begin{aligned} \phi \;&=\; p \mid \neg\phi \mid \phi \wedge \phi && \text{Boolean Modalities with } p \in \mathcal{AP} \\ &\quad\; X\phi \mid \phi U \phi && \text{Future Temporal Modalities} \end{aligned}$$

- $X\phi$ is the Next modality: *at the next time point (tomorrow), the formula ϕ holds*
- $\phi_1 U \phi_2$ is the Until modality : *there exists a time point in the future where ϕ_2 is true, and ϕ_1 holds from now until (but not necessarily including) that point.*

Shortcuts:

- Eventually $F\phi$: *there exists a time point in the future where ϕ holds.*
  It is defined as $F\phi = \top U \phi$.
- Globally $G\phi$: *for all time points in the future ϕ holds.*
  It is defined as $G\phi = \neg F \neg \phi$.

Linear Temporal Logic with Past (LTLP)

The syntax of LTLP is defined as follows:

$$\begin{aligned} \phi \;=\; & p \mid \neg\phi \mid \phi \wedge \phi && \text{Boolean Modalities with } p \in \mathcal{AP} \\ & X\phi \mid \phi U \phi && \text{Future Temporal Modalities} \\ & Y\phi \mid \phi S \phi && \text{Past Temporal Modalities} \end{aligned}$$

- $Y\phi$ is the Yesterday modality: *the previous time point exists and it satisfies* ϕ.
- $\phi_1 S \phi_2$ is the Since modality: *there exists a time point in the past where* ϕ_2 *is true, and* ϕ_1 *holds since (and excluding) that point up to now.*

Shortcuts:

- Once $O\phi$: *there exists a time point in the past where* ϕ *holds.* $O\phi = \top S \phi$.
- Historically $H\phi$: *for all time points in the past* ϕ *holds.* $H\phi = \neg O \neg \phi$.
- Weak yesterday $\tilde{Y}\phi$: $\tilde{Y}\phi = \neg Y \neg \phi$

The cosafety fragment of LTL

We say that a temporal logic $\mathbb{L}$ is *cosafety* iff, for any $\phi \in \mathbb{L}$, $\mathcal{L}(\phi)$ is *cosafety*.

cosafetyLTL

Definition

$\phi := p \mid \neg p \mid \phi \vee \phi \mid \phi \wedge \phi \mid X\phi \mid F\phi \mid \phi U \phi$

Example:

pUq

F(pLTL)

Definition

$\phi := F(\alpha)$, where α is a formula of the pure-past fragment of LTLP (pLTL).

Example:

$F(q \wedge \tilde{Y}Hp)$

F(pLTL) is the canonical form of cosafetyLTL.

The safety fragment of LTL

We say that a temporal logic $\mathbb{L}$ is *safety* iff, for any $\phi \in \mathbb{L}$, $\mathcal{L}(\phi)$ is *safety*.

safetyLTL

Definition

$\phi := p \mid \neg p \mid \phi \vee \phi \mid \phi \wedge \phi \mid X\phi \mid G\phi \mid \phi R \phi$

Example:

$G(r- > XXg)$

G(pLTL)

Definition

$\phi := G(\alpha)$, where α is a formula of the pure-past fragment of LTLP (pLTL).

Example:

$G(\tilde{Y}\tilde{Y}r- > g)$

G(pLTL) is the canonical form of safetyLTL.

Monitorability behind safety and cosafety fragments / 1

The class of monitorable LTL properties is larger than the union of safety and cosafety properties.

$$((p \vee q)Ur) \vee Gp$$

On the one hand, observe that:

- $ppp\ldots$ satisfies the formula (but none of its prefixes is good - not a cosafety property)
- $qqq\ldots$ does not satisfy the formula (but none of its prefixes is bad - not a safety property)

On the other hand:

- $\ldots r$ is a good prefix for the formula, provided that one of p or q holds in positions denoted by $\ldots$
- $\ldots \{\neg p, \neg q, \neg r\}$ is a bad prefix for the formula

Key point. Any finite prefix that is neither good nor bad can be extended to a good or a bad prefix: any letter containing r makes the prefix good, while a continuation with the letter $\{\neg p, \neg q, \neg r\}$ makes the prefix bad.

Properties which are not monitorable

There are properties which are neither positively nor negatively monitorable.

This is, for instance, the case with the property (*reactivity* property):

Every request is sooner or later granted

- If a request has not been yet granted, you cannot exclude that it will be granted in the future.
- If up to now all requests have been granted, you cannot exclude that a future one will not.

Limitations of monitoring

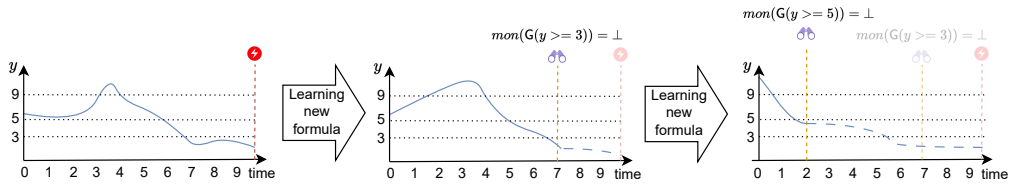
Monitoring suffers from some significant limitations.

Among them we would like to mention the following ones:

- modern systems have such a level of complexity that it is impossible for a system engineer to specify in advance all properties to be monitored;
- even minor changes to the system can introduce unforeseen bugs.

Learning what to monitor

How to solve this problem? By pairing monitoring and learning: **learn**, in an online, iterative fashion, **new formulas to be monitored** against the system by analysing trace prefixes that lead to failure events.



Given a trace, generate a formula that triggers only on the failing prefix, not the good part (i.e., contrastive fashion).

Pairing monitoring with machine learning

- system engineers specify only a set of initial properties.
- *Warmup (offline) Phase*: in a fully automatic fashion, the framework analyzes the traces of the system that lead to a failure, and derive new relevant properties, with the objective of *anticipate* their identification.
- *Online Phase*: the framework monitors the system in real time. If a failure occurs, it derives new relevant formulas and it iteratively refines the pool of formulas to be monitored.

A proof of concept with Signal Temporal Logic and Genetic Programming.

From LTL to STL (Signal Temporal Logic)

STL (Signal Temporal Logic) extends LTL by pairing its (qualitative) semantics with a quantitative one.

The qualitative semantics of STL determines whether a signal satisfies a given formula or not.

The quantitative semantics of assigns a real-valued measure reflecting the degree of satisfaction or violation (robustness).

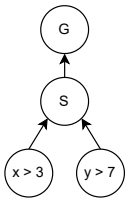
Example. $G_{[0.5,3.75]}(\mathit{water.temperature} > 50 \wedge \mathit{water.pressure} \leq 2.5)$

Safety and cosafety fragments of STL can be defined as in the case of LTL, and they admit similar characterizations.

Genetic programming task

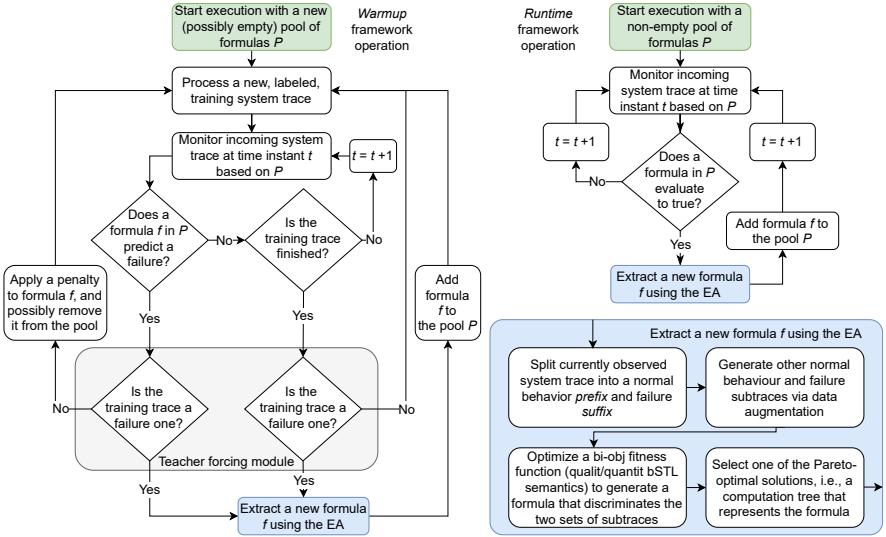
Input: failure trace prefix $tr[:(t-1)]$ and good training traces set T_g

- Generate augmented traces T_{aug} from tr
- Evolutionary part:
  - Each individual is a formula ϕ (represented by its syntax tree)
  - Mutation and crossover operators act on the syntax trees
  - Three-fold fitness function:
    1. effectiveness of ϕ in recognizing T_{aug} traces as failing ones
    2. while avoiding false positives on T_g
    3. award early identification wrt time instant $(t-1)$
  - At the end, get the formula with the highest hypervolume from the Pareto front



Output: formula ϕ expressed in pSTL, monitorable by construction

The framework



Distinguishing features of the framework:

- *interpretability*: the machine learning methods manipulate and produce only formulae, that can be easily inspected by a system engineer;
- formal guarantees on *monitorability*: every formula produced during the learning phase is guaranteed to be monitorable (this is done syntactically, through the grammar used for the generation of the computation tree of each formula);
- *generality*: different monitoring and machine learning backends.

Limitations of "pure" machine learning approaches

The machine learning techniques that are most used in predictive maintenance and early failure detection are:

- Random Forests
- Artificial Neural Networks
- Support Vector Machines

The *lack of interpretability* is a problem common to all of the above techniques, in the sense that they fail to provide an explanation of their prediction.

Explanation of the output of a predictive maintenance algorithm is important not only for humans trying to understand the error but also for implementing the correct actions for preventing the failure of the asset under consideration.

Extensions to the framework

The framework can be extended along the following directions:

- *automata-less monitoring via trace checking* for intentionally safe and cosafe formulas (from doubly-exponential to polynomial complexity)
- identification of *anomalies* and *drops of performance*
- exploitation of *unsupervised and self-supervised learning* techniques
- modularity

Automata-less monitoring via trace checking

The notions of *intentionally safe* and *intentionally cosafe* formulas are based on the concept of informative models.

A model is informative for a formula ϕ if it contains sufficient information to determine whether ϕ is true or false.

Example. The word $< \{p\} >$ is an informative model for the LTL formula $F(p)$, since p holds at the first position, but it is *not* informative for $F(p \wedge (Xq \vee X\neg q))$, because evaluating the formula requires a position satisfying p followed by a position where either q or $\neg q$ holds, but $< \{p\} >$ has no successor.

For all intentionally cosafe (resp., safe) LTL formulas, monitoring can be performed in an *automata-less* fashion: it can be reduced to checking whether the formula is satisfied (resp., violated) by the current trace (*trace-checking* problem).

Anomaly detection and self-supervised learning backend

- The framework currently works in a **supervised fashion**
  - traces, labeled as failure or good behaviour ones, guide the first (*warmup*) stage of formula extraction, following a teacher-forcing like approach
  - justified as failures are terminating events (always detectable)
- The new version shall be **self-supervised** to deal with anomalies:
  - characterizing *a priori* anomalies in modern complex systems is impractical
  - systems evolve continuously over time
  - supervised assumption, i.e., existence of complete and exhaustive dataset of labelled anomalies is unrealistic and unfeasible
- Possible solution: use deep learning approaches capable to perform self-supervised anomaly detection as a source of supervision

Assumption-based runtime verification

Assumption-Based Runtime Verification (ABRV) has recently been introduced as a variant of monitoring to deal with systems that are only *partially observable*

- classical monitoring restricts itself to observable parts of the system and treat the non-observable ones as *black boxes*
- ABRV exploits the fact that in practice one always knows something about the internal (non-observable) parts of the system in form of *assumptions* that the domain expert can specify before monitoring
- ABRV can reach conclusive verdicts with shorter trace prefixes

Modularity

The new framework shall be *modular* in at least the following dimensions:

- the specification language
  - different temporal logics including LTL, STL, and ITL
  - *qualitative* semantics (for tasks like failure detection)
  - *quantitative* semantics (for tasks like anomaly detection), where appropriate
- the backend implementing the monitoring algorithm
- the backend for the learning of new properties
  - move across learning paradigms and tasks
  - different solution than GP for formula extraction (which is limited by bloat, huge search space, tree-based formula representation, etc.), like the integration with reinforcement learning or generative AI; alternatively, formulas can be represented as graphs, enabling the usage of Graph Neural Networks

Some references - 1

- Bauer, A., Leucker, M., & Schallhart, C. (2011). Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology.
- Cimatti, A., Tian, C., & Tonetta, S. (2022). Assumption-based Runtime Verification. Formal Methods in System Design.
- Geatti L. & Montanari, A. (2023). The Safety Fragment of Temporal Logics of Infinite Sequences . ESSLLI Summer School
- Brunello, A., Della Monica, D., Montanari, A., Saccomanno, N., & Urgolo, A. (2023). Monitors that Learn from Failures: Pairing STL and Genetic Programming. IEEE Access.
- Geatti L., Montanari, A., & Saccomanno, N. (2023). Towards Machine Learning Enhanced LTL Monitoring. OVERLAY workshop

Some references - 2

- Brunello, A., Geatti, L, Montanari, A., & Saccomanno, N., (2024). Learning what to Monitor: using Machine Learning to Improve Past STL Monitoring. Proceedings of the 33rd International Joint Conference on Artificial Intelligence (IJCAI), Jeju Island, South Korea.
- Brunello, A., Geatti, L, Montanari, A., & Saccomanno, N. (2025). Interpretable Early Failure Detection via Machine Learning and Trace Checking-based Monitoring. Proceedings of the 28th European Conference on Artificial Intelligence (ECAI), Bologna, Italy.
- Brunello, A., Geatti, L, Montanari, A., & Saccomanno, N. (2025). Automata-less Monitoring via Trace-Checking. Submitted for publication.