Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Monitors that Learn from Failures: Pairing STL and Genetic Programming

ANDREA BRUNELLO[1], DARIO DELLA MONICA[1], ANGELO MONTANARI[1], NICOLA SACCOMANNO[1], ANDREA URGOLO[1]

[1]Dept. of Mathematics, Computer Science, and Physics, University of Udine (e-mail: {andrea.brunello | dario.dellamonica | angelo.montanari | nicola.saccomanno | andrea.urgolo}@uniud.it).

Corresponding author: Andrea Urgolo (e-mail: andrea.urgolo@uniud.it).

This project was done within the COMET Centre ASSIC (Austrian Smart Systems Integration Research Center), which is funded by BMK, BMDW, and the Austrian provinces of Carinthia and Styria, in the COMET (Competence Centers for Excellent Technologies) framework. The COMET programme is run by FFG. The authors also acknowledge the support from the 2022 Italian INdAM-GNCS project *Elaborazione del Linguaggio Naturale e Logica Temporale per la Formalizzazione di Testi* CUP_E55F22000270001.

ABSTRACT In several domains, systems generate continuous streams of data during their execution, including meaningful telemetry information, that can be used to perform tasks like preemptive failure detection. Deep learning models have been exploited for these tasks with increasing success, but they hardly provide guarantees over their execution, a problem which is exacerbated by their lack of interpretability. In many critical contexts, formal methods, which ensure the correct behaviour of a system, are thus necessary. 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. To overcome these limitations, we design a framework that resorts to monitoring, a lightweight runtime verification technique that does not require an explicit model specification, and pairs it with machine learning. Its goal is to automatically derive relevant properties, related to a bad behaviour of the considered system, encoded by means of formulas of Signal Temporal Logic (STL). Results based on experiments performed on well-known benchmark datasets show that the proposed framework is able to effectively anticipate critical system behaviours in an online setting, providing human-interpretable results.

INDEX TERMS Machine Learning, Formal Methods, Runtime Verification, Monitoring, Failure Detection, Explainable AI.

I. INTRODUCTION

In many application domains, during its operation a system produces several data streams, that may contain valuable telemetry information. This is the case, for instance, with the logs generated by web servers, smart sensors, and industrial machinery. Such data can be used for tasks like predictive maintenance and early failure detection, typically carried out, due to their complexity, by means of machine or deep learning approaches. Despite their success, these methods hardly provide any guarantees over their execution, a problem which is exacerbated by their lack of interpretability, which is an essential requirement in many critical domains, such as, for instance, healthcare and avionics. In those scenarios, formal methods can, in principle, be used to automatically verify software and hardware systems. However, the presence of different operating conditions combined with the complexity of the system components and their interactions make it quite difficult to define in advance all the relevant conditions that must be guaranteed (or avoided) during execution; moreover, the specification of a complete system model against which to check these properties may be simply impossible [1].

To overcome these limitations, various methods, that combine classic exhaustive formal verification techniques with model-based testing and monitoring, have recently been proposed in the literature (see, e.g., [2], [3]). Here, we focus on the latter. *Monitoring* [1] is a runtime verification technique which is receiving more and more attention from the formal verification community. It allows one to detect the fulfilment or violation of a property (usually expressed by a temporal logic formula) by evaluating a single system run, without requiring a model of the system being considered. This makes it naturally applicable to data streaming scenarios.

In this paper, we present a novel online system verification framework that combines monitoring with supervised

machine learning and can be used for tasks like preemptive failure detection over streams of data. The framework starts its operation by considering a limited set of properties encoding bad behaviours, to be monitored against the system, learnt during a *warmup* phase and/or specified with the help of domain experts. Then, during the *runtime* phase, by means of an iterative refinement process, the framework autonomously discovers new relevant properties, becoming able, over time, to identify undesired behaviours in advance, and with a significantly higher level of detail and coverage than the initial specifications. The process of property discovery and extraction is carried out by means of an original bi-objective evolutionary algorithm.

The distinctive features of the proposed solution are the following:

- the framework poses as a monitoring-based tool to perform preemptive failure detection;
- its operation relies on the seamless and automatic interaction between formal methods and machine learning approaches;
- thanks to its modularity and flexibility, the framework can be adapted to different application domains and contexts; in particular, Signal Temporal Logic (STL) can possibly be replaced by other logical formalisms for property specification;
- interpretability is a distinguishing feature of the framework, as the produced responses can be easily read by domain experts; this allows people to validate the overall behaviour of the framework, and to gain insights about the causes that led to a failure;
- the framework works in an online fashion, and it can adapt to changes in the behaviour of the system, due, for instance, to updates or upgrades.

The framework has been evaluated against three public datasets, and it has shown to be able of actually predicting in advance system failures. Results are on par with those obtained from other state-of-the-art solutions that, however, suffer from a lack of interpretability.

The rest of the paper is organized as follows. Section II analyses related work. Section III provides background knowledge about monitoring, STL, and evolutionary algorithms. Section IV describes the implementation of the evolutionary algorithm used in the property extraction phase. Section V shows how such an algorithm has been incorporated in the proposed framework. The experimental evaluation of the framework is reported in Section VI. Section VII provides a critical assessment of the work done and discusses its strengths and limitations. The last section summarizes the main contributions and outlines future research directions.

II. RELATED WORK

Learning techniques for the real-time detection of undesired behaviours (failures) of complex systems are getting increasingly popular. A significant line of research makes use of machine learning and deep learning, that realize failure detection via black-box models rather than by providing explicit properties capable of characterizing bad behaviours of a system. Despite their lack of interpretability, that makes it difficult to understand and validate the resulting verdicts, these approaches have been employed in several domains due to their effectiveness. For instance, machine learning strategies based on Logistic Regression (LR), Support Vector Machine (SMV), Random Forest (RF), and K-Nearest Neighbours (KNN) applied to the domains of aircraft components post-flight reports, gearbox failures in industrial robots, high-performance computing, and cloud systems are described in [4]–[6]. Deep learning solutions are typically exploited to extract temporal relations in time series data, as witnessed in [7]–[10], where Long Short-Term Memory (LSTM) and Recurrent Neural Networks (RNNs) are applied to the domains of job failures in large-scale cloud data centres, turbofan engine degradation, hard drive telemetry data, and heart atrial fibrillation detection on routine screening electrocardiogram (ECG) signals. A common limitation of all these solutions is that historical data used for the predictions are often defined through a time window of fixed size, which may be inadequate when heterogeneous failure behaviours have to be captured.

In the attempt to cope with the interpretability requirement, which is fundamental in many critical domains, some approaches for the extraction of properties that distinguish between failure and normal execution traces of a system have been recently proposed in the literature [11]–[17]. However, learning temporal properties from the observed system traces is a challenging task that involves intractable optimization problems [11]. To overcome them, heuristics were suggested [11]–[14]. In this spirit, some ad hoc, domain-specific solutions have been devised to assess the condition of electrical rotating machines through real-time vibration measurement and analysis instruments [18], to discover temporally-constrained alarm sets from dynamic systems' logs [19], to diagnose rolling bearings faults through a hardware architecture with a reconfigurable logic based on field programmable gate arrays (FPGAs) [20], to detect system intrusions through temporal logic specifications [21], and to ensure the safety of synthesized policies for robotics through model predictive shielding [22]. Nonetheless, a general technique, applicable to different domains and contexts, is still missing. A step towards such a goal was taken in [23], where an STL-based solution to the problem of detecting ineffective respiratory effort in intensive care patients, which includes a learning phase supporting some adaptive behaviours, is outlined. Still, the generative data models employed in the learning phase are tailor-made, thus limiting the flexibility and generality of the proposed solution.

With the goal of generalizability in mind, approaches explicitly aimed at combining the points of strength of machine learning and formal methods have been recently proposed. Specifically, in [15]–[17], techniques for the mining of STL properties that distinguish between two different sets of time series data are presented. The proposal in [15] relies on a genetic algorithm combined with parameter learning through

Gaussian process confidence upper bound. The solution originally presented in [13], and then extended with online learning in [16], exploits a decision tree learner based on STL primitives. Finally, the approach in [17] relies on a reinforcement learning-based property extractor that combines data- and knowledge-driven methodologies. Still, the aforementioned proposals significantly differ from what is presented in this work, as they are not designed to work iteratively, managing a pool of properties in real-time. Beyond STL, in [24], a failure prediction method for cloud data centres, based on message pattern recognition via Bayesian probability, is described. As for failure detection in cyber-physical systems, a Ripple down rule-based (RDR) framework was proposed in [25], that exploits a machine learning technique based on the algorithm InductRDR [26]; the result is then maintained by domain experts, who refine the RDR knowledge base.

A related field is that of specification mining, whose goal is to generate/integrate the formal specification of a system by analyzing its execution traces. Various approaches to this problem have been proposed in the literature. In [27], a Linear Temporal Logic (LTL) property template miner, based on support and confidence thresholds, is devised. A Bayesian inference-based probabilistic model that generates LTL task specifications from examples, by exploiting a Markov Chain Monte Carlo algorithm, is outlined in [28]. In [29], an algorithm to infer LTL specifications by combining the representational power and interpretability of temporal logic with the generalizability of inverse reinforcement learning is proposed. The problem of mining finite state automata to generate formal specifications in the context of software applications and libraries is dealt with in [30]. To this end, the authors make use of prefix tree acceptors, language models based on recurrent neural networks, and clustering algorithms to merge similar automaton states. Despite the relevance of specification mining, the proposed solutions extract non-contrastive properties from the observed executions of a system, and thus they cannot be naturally applied to failure detection, where properties able to discriminate between good and bad behaviours are needed.

As a final remark, we would like to mention that the solution described in detail in this paper was first outlined in two short preliminary contributions [31], [32]. In this paper, we fully work out the proposed framework, revise and extend it in various ways with respect to its original formulation, and largely improve its experimental evaluation.

III. BACKGROUND KNOWLEDGE

In this section, we recall some basic notions about monitoring, STL, and evolutionary algorithms.

A. MONITORING

While classic verification techniques, like, for instance, model checking [33], perform an exhaustive analysis of the behaviours of a system, monitoring [1] aims at establishing satisfaction or violation of a property by analyzing a finite prefix of a single behaviour (*trace*/*run*), and then issuing an irrevocable verdict [1]. It is thus a lightweight technique, but the gain in efficiency is paid in terms of expressivity: monitorable properties are a subset of those expressible in temporal logics commonly used for automated verification.

We say that a property is *positively* (resp., *negatively*) *monitorable* if every trace satisfying (resp., violating) it features a finite prefix that witnesses the satisfaction (resp., violation). A *monitorable* property is a property that is either positively or negatively monitorable. Safety properties, informally requiring that "*something bad will never happen*", are negatively monitorable, as their violation is witnessed by a finite prefix exhibiting a violation; dually, co-safety properties, stating that "*something required will eventually happen*", are positively monitorable. Notably, there are meaningful properties, like, e.g., "*a good state is accessed infinitely often*" (an essential ingredient of strong fairness requirements [34]), which are clearly neither positively nor negatively monitorable.

As we will see in Section V, the online nature of monitoring makes it a natural candidate for the proposed framework.

B. SIGNAL TEMPORAL LOGIC (STL)

Signal Temporal Logic (STL) [35] extends propositional logic with future modalities that allow one to express temporal properties over linear structures. It can be directly applied to time series data characterized by continuous values.

Let $\mathbb{N}_{>0}$ (resp., $\mathbb{N}_{[t,t']}$) be the set of positive naturals (resp., naturals in between t and t', for all $t, t' \in \mathbb{N}$) and $\mathbb{R}^n$ be the n-dimensional Euclidean space over reals. A discrete-time STL *signal* (or *trace*) is a function $x : \mathbb{N} \to \mathbb{R}^n$, for some $n \in \mathbb{N}_{>0}$; a *partial signal* is a function $x : \mathbb{N}_{[0,t]} \to \mathbb{R}^n$, for some $n \in \mathbb{N}_{>0}$ and $t \in \mathbb{N}$.[1] We denote the *length* of a (partial) signal x by $len(x)$. For a signal x, it holds that $len(x) = \infty$, whereas $len(x) = t + 1$ for a partial signal $x : \mathbb{N}_{[0,t]} \to \mathbb{R}^n$. Let $\mathcal{X}$ (resp., $\tilde{\mathcal{X}}$) be the set of signals (resp., partial signals). If the codomain of a (partial) signal x is $\mathbb{R}^n$, then n is the *dimension* of x, denoted by $|x|$. Let $x \in \mathcal{X} \cup \tilde{\mathcal{X}}$. We denote by x_i, with $1 \leq i \leq |x|$, the function from the domain of x to $\mathbb{R}$ such that $x_i(t)$ is equal to the i-th component of $x(t)$, for all t. Moreover, we denote by $x[j, k]$, with $0 \leq j \leq k < len(x)$, the restriction of the function x to the domain $\mathbb{N}_{[j,k]}$.

The syntax of STL is given by the grammar:

$$\phi ::= \top \mid x_i \geq c \mid \neg\phi_1 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathsf{U}_I \phi_2,$$

where x is a signal, $1 \leq i \leq |x|$, $c \in \mathbb{R}$, and I is an interval of the form (a, b), $(a, b]$, $[a, b)$, or $[a, b]$, with $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \{\infty\}$, and $a \leq b$. Modality U (*until*) is paired with an interval I which defines its validity scope. For every $t \in \mathbb{N}$ and interval $I = (a, b)$, we denote by $t + I$ the interval $(t + a, t + b)$ (the same for intervals of the forms $(a, b]$, $[a, b)$, and $[a, b]$). Derived modalities are defined as usual. As an

[1] As a matter of fact, STL allows one to deal with continuous-time signals by simply redefining x as $x : \mathbb{R}_{\geq 0} \to \mathbb{R}^n$, where $\mathbb{R}_{\geq 0}$ is the set of non-negative reals. Here, we restrict ourselves to the discrete-time case given that a sampling is required to represent time series data within a dataset.

example, modalities *eventually* and *globally* are defined as $\mathsf{F}_I\phi = \top \mathsf{U}_I \phi$ and $\mathsf{G}_I\phi = \neg\mathsf{F}_I\neg\phi$, respectively.

STL pairs the standard Boolean semantics with a quantitative one, which measures the *robustness* of the satisfaction of ϕ by a signal x at a time $t \in \mathbb{N}$.

The quantitative semantics of STL is inductively defined as follows:

- $\rho(\top, x, t) = +\infty$;
- $\rho(x_i \geq c, x, t) = x_i(t) - c$;
- $\rho(\neg\phi, x, t) = -\rho(\phi, x, t)$;
- $\rho(\phi_1 \wedge \phi_2, x, t) = \min\{\rho(\phi_1, x, t),\ \rho(\phi_2, x, t)\}$;
- $\rho(\phi_1 \mathsf{U}_I \phi_2, x, t) = \max_{t_1 \in t+I} \min\{\rho(\phi_2, x, t_1), \min_{t_2 \in [t,t_1)} \rho(\phi_1, x, t_2)\}$.

The Boolean semantics of STL is defined on the basis of the sign of $\rho(\phi, x, t)$:

$$x, t \models \phi \text{ if and only if } \rho(\phi, x, t) \geq 0.$$

Finally, given a partial signal $x \in \bar{\mathcal{X}}$ over $\mathbb{N}_{[0,t]}$, the set of *completions* of x is defined as $\mathcal{C}(x) = \{\hat{x} \in \mathcal{X} \mid \hat{x}(t') = x(t') \text{ for all } t' \in \mathbb{N}_{[0,t]}\}$.

STL monitoring is formally defined by the function $mon : \bar{\mathcal{X}} \times \mathsf{STL} \to \{\top, \bot, ?\}$ such that $mon(x, \varphi)$ returns $\top$ iff $\hat{x}, 0 \models \varphi$ for all $\hat{x} \in \mathcal{C}(x)$ iff $\rho(\varphi, \hat{x}, 0) \geq 0$ for all $\hat{x} \in \mathcal{C}(x)$; $\bot$ iff $\hat{x}, 0 \not\models \varphi$ for all $\hat{x} \in \mathcal{C}(x)$ iff $\rho(\varphi, \hat{x}, 0) < 0$ for all $\hat{x} \in \mathcal{C}(x)$; ? otherwise.

The choice of STL as the formalism for the specification of the properties to monitor has various advantages. First of all, STL allows one to directly deal with real values, still featuring quite compact and interpretable formulas. Moreover, its quantitative semantics provides one with an additional tool to evaluate the behaviour of the extracted formulas, a feature that will be described in detail in Section IV.

C. MONITORING BOUNDED STL FORMULAS

Monitoring properties that refer to both the current and the future behaviour of a system is a challenging task since their evaluation at a given time t may also depend on the observed inputs at some time $t' > t$. In Section III-A and Section III-B, we introduced the notion of monitorable properties and provided syntax and semantics of STL, respectively. To the best of our knowledge, no tool supporting the monitoring of arbitrary STL formulas is available. Luckily, in most application domains, the properties to monitor can be expressed by means of bounded-time STL (bSTL) formulas, and a tool to deal with such a class of formulas has been developed in [36]. Basically, the fragment bSTL constrains the interval I associated with modality U to be finite, that is, $I = [a, b]$, with both a and b belonging to $\mathbb{N}$ ($b = \infty$ is excluded).

Let ϕ be a bSTL formula. By analyzing its syntactic structure, one can compute a temporal *horizon* $H(\phi)$, that intuitively represents the maximum number of (future) time points that one must take into consideration to establish whether or not ϕ is true. In the following, when evaluating a bSTL formula ϕ, with temporal horizon $H(\phi)$, at a given time t, the monitor will wait until time $t + H(\phi)$ is reached, since at that time all the data necessary for the quantitative evaluation of the formula and the possible formulation of a $\top$ or $\bot$ verdict have surely been observed. As an example, the horizon of the bSTL formula $\phi = x \geq 3\ \mathsf{U}_{[0,3]}\ x \geq 5$ is 3, and thus only after 3 time units we can complete its (quantitative) evaluation.

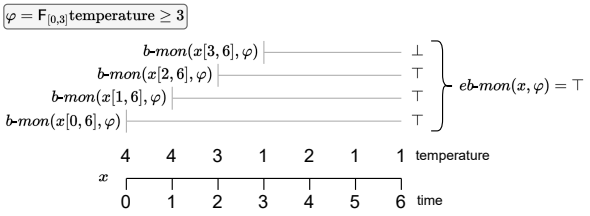


FIGURE 1. $eb\text{-}mon$ run on a partial signal x. The execution of $b\text{-}mon$ over all the suffixes of x longer than the horizon is also reported.

Formally, the temporal *horizon* $H(\phi)$ of a bSTL formula ϕ is defined as follows:

- $H(\top) = 0$;
- $H(x_i \geq c) = 0$;
- $H(\neg\phi) = H(\phi)$;
- $H(\phi_1 \wedge \phi_2) = max\{H(\phi_1), H(\phi_2)\}$;
- $H(\phi_1\ \mathsf{U}_{[a,b]}\phi_2) = b + max\{H(\phi_1) - 1, H(\phi_2)\}$.

Notice that the monitor, when applied to a bSTL formula ϕ, may output the truth values $\top$ or $\bot$, as well as the undefined verdict ? when the horizon of ϕ still has to be reached. As previously mentioned, a $\top$ or $\bot$ can always be reached when the horizon is met.

Formally, bSTL monitoring is defined by the function $b\text{-}mon : \bar{\mathcal{X}} \times \mathsf{bSTL} \to \{\top, \bot, ?\}$ such that $b\text{-}mon(x, \varphi)$ returns $\top$ if $mon(x, \varphi)$ returns $\top$ and $len(x) \geq H(\varphi)+1$; $\bot$ if $mon(x, \varphi)$ returns $\bot$ and $len(x) \geq H(\varphi)+1$; ? otherwise.

Now, we observe that, by the definition of monitoring and the nature of bSTL formulas, monitors evaluate bSTL formulas only based on prefixes of signals of bounded length, the bound depending on the temporal horizon of the formulas. As an example, formula $\varphi = \mathsf{F}_{[0,3]}\ temperature \geq 3$ states that there must exist at least one time point where $temperature \geq 3$ among the first 4 (i.e., $H(\varphi) + 1$) time points of the signal, that is, in the set of time points $\{0, 1, 2, 3\}$. This limits the applicability of monitoring in real-world scenarios where one is interested in detecting the possible occurrence of a given condition at any time point of a signal. This is the case, for instance, with the property: *"in 25 time units from now the temperature will exceed 30 degrees"*. To accommodate for that, we extend the notion of monitoring by making it possible to apply it to any time point, that is, to any suffix of a signal. To this end, building upon function $b\text{-}mon$, we define function $eb\text{-}mon : \bar{\mathcal{X}} \times \mathsf{bSTL} \to \{\top, \bot, ?\}$ such that:

$$eb\text{-}mon(x, \varphi) = \begin{cases} ? & \text{if } len(x) < H(\varphi) + 1 \\ \bigvee\limits_{0 \leq i \leq len(x)-1-H(\varphi)} b\text{-}mon(x[i, len(x) - 1], \varphi) & \\ & \text{otherwise} \end{cases}$$

We are interested in identifying signals exhibiting bad behaviours, which are encoded by means of bSTL properties. As noted before, a conclusive verdict can be issued for a signal only if it is longer than the horizon of the bSTL property under consideration. Therefore, given a partial signal x and a bSTL property φ, function $eb\text{-}mon(x, \varphi)$ returns ? whenever x is no longer than the horizon of φ. Otherwise, we monitor (through $b\text{-}mon$) φ against all suffixes y of x longer $H(\varphi)$: if at least one such monitoring procedures returns $\top$, then so does $eb\text{-}mon(x, \varphi)$, meaning that the signal is considered a bad-behaving one (see Figure 1).

The monitoring tool we used to realize the above-described approach is `rtamt`, a Python library for monitoring discrete- and dense-time bSTL properties [36].

D. EVOLUTIONARY ALGORITHMS

Evolutionary Algorithms (EAs) are population-based metaheuristics inspired by the process of biological evolution and genetics, that excel in the solution of combinatorial optimization problems [37]. Unlike classic random search, EAs make use of historical information to direct the search into the most promising regions of the search space.

In nature, a *population* of individuals tends to evolve to adapt to its environment. Similarly, EAs are characterized by a population, where each individual represents a candidate solution to a given optimization problem; each solution is evaluated with respect to its degree of "adaptation" to the problem through a single- or multi-objective *fitness* function.

The EA population iteratively goes through a series of *generations*. At each generation, individuals chosen by a *selection* strategy undergo a process of reproduction. Such a selection strategy is the fundamental factor that distinguishes one evolutionary based approach from another, although, typically, individuals with a high degree of adaptation are more likely to be chosen (*elitism*). In this way, the elements of the population progressively evolve toward better solutions. Reproduction involves the application, with a certain degree of probability, of suitable *crossover* and *mutation* operators. As a result, an offspring is generated, which is finally merged with the previous population, and the cycle repeats until a stopping condition is met, e.g., a condition based on a given fitness threshold.

Crossover is the EA counterpart of natural reproduction, by which the characteristics of two individuals are combined by generating one or two offspring. As a general rule, a high crossover probability tends to pull the population towards a local minimum or maximum. Mutation applies random changes to the encoding of the selected solution, with the goal of maintaining genetic diversity in the individuals; it prevents premature convergence of the algorithm to a local optimum, thus allowing it to explore the search space more broadly.

In this work, we deal with a specific kind of optimization task, that is, *genetic programming*. Such a technique evolves programs starting from a population of random solutions [38]. Each individual is encoded by means of a computation tree, where each leaf represents an input value (either a variable or a constant) and internal nodes encode operators. The output value is generated by the primitive encoded in the root. Typical crossover/mutation operations applied on computation trees are subtree exchange and node/leaf addition/removal/replacement.

As for the task of property extraction, we will rely on a multi-objective evolutionary algorithm. The reason is twofold. First, such an EA is able to simultaneously follow different optimization goals, producing a set of Pareto-optimal solutions as a result which one can subsequently combine. Second, it is a flexible approach, as it allows one to customize the syntax of the generated formulas by constraining the computation trees, e.g., enabling/disabling specific operators or allowing only some kinds of combinations among them.

IV. THE EVOLUTIONARY ALGORITHM

The Evolutionary Algorithm we are going to exploit relies on DEAP (Distributed Evolutionary Algorithms in Python) [39], a framework that provides practical tools for the prototyping of custom evolutionary algorithms. In this section, we will illustrate how the different components of the optimizer have been developed.

The algorithm receives a set of finite traces $\mathcal{X}$ (all of the same length) as input, then, it partitions each trace into a normal behaviour prefix and a failure behaviour suffix, and, finally, it generates a bSTL formula which is able of discerning between the two cases.

A. POPULATION AND ITS INITIALIZATION

Each individual belonging to the population consists of a pair (φ, w), where φ encodes a computation tree representing a syntactically correct bSTL formula and w is its associated *normal behaviour window length*.

As for φ, it is generated following DEAP's *genHalfAndHalf* method, which outputs a random tree with a maximum height of 6, as suggested by Koza in his seminal work [40]. More precisely, half the time a tree whose leaves have all the same depth is returned; in the remaining cases, different leaves may have different depths.

The window length w is used to partition each trace $x \in \mathcal{X}$ into a normal behaviour prefix of length w and failure suffix of length $len(x) - w$ (see the definition of the fitness function below). Note that, in the generation process of the individual, a formula φ with a horizon $H(\varphi) \geq len(x) - w$ might be obtained. In such a case, the individual is discarded and another one is generated. The process is iterated until a valid individual is obtained.

B. NODES OF THE COMPUTATION TREE

A node of the computation tree may represent a constraint, e.g., $x_i \geq c$, a bSTL formula whose outermost operator is a temporal one, e.g., $\varphi \mathsf{U}_{[a,b]} \psi$, or a Boolean formula like, for instance, $\varphi \vee \psi$, where φ and ψ are bSTL sub-formulas which are represented, in their turn, as trees. A node may also encode the following terminal values: (i) interval bounds of

a temporal operator, i.e., $[a, b]$, with $a, b \in \mathbb{N}$ and $a \leq b$, (ii) signal identifiers x_i, with $1 \leq i \leq |x|$, and (iii) constants c occurring in formulas, with $c \in Dom(x_i)$ for some i. All these terminals are implemented by means of DEAP's *EphemeralConstants*. As for the length of the normal behaviour window, it is implemented as an *EphemeralConstant* w, with $0 < w < len(x)$, where $len(x)$ is the length of the traces $x \in \mathcal{X}$ (they are all of the same length).

C. FITNESS FUNCTION

In order to evaluate an individual of the population, each trace $x \in \mathcal{X}$ is logically partitioned into a good behaviour prefix $x[0, w-1]$ and a failure suffix $x[w, len(x)-1]$, following a windowing approach which takes w as the length of the normal behaviour window. A bi-objective fitness function is then defined by making use of the `rtamt` monitoring algorithm for bSTL.

Formally, the first objective measures how good a formula φ is in discriminating between the normal behaviour prefixes and the failure suffixes. For each trace x and each formula φ, let us define the numerical counterpart of $eb\text{-}mon(x, \varphi)$ as follows:

$$NUM(eb\text{-}mon(x,\varphi)) = \begin{cases} 1 & \text{if } eb\text{-}mon(x,\varphi) = \top, \\ 0 & \text{otherwise.} \end{cases}$$

The first objective measure is defined as follows:

$$Acc(\mathcal{X},\varphi) = \frac{\sum_{x\in\mathcal{X}} 1 - NUM(eb\text{-}mon(x[0, w-1+H(\varphi)], \varphi)) + \sum_{x\in\mathcal{X}} NUM(eb\text{-}mon(x[w, len(x)-1], \varphi))}{2 \cdot |\mathcal{X}|},$$

It is worth noticing that, to maximize $Acc(\mathcal{X}, \varphi)$, a formula φ should evaluate to $\bot$ on the normal behaviour prefixes and to $\top$ on the failure suffixes. In this respect, it is very important to be able to evaluate a formula φ to $\top$ or $\bot$ till the last instant of the good behaviour prefix of a trace x. To this end, we simply extend the prefix with the first $H(\varphi)$ points taken from the failure suffix. Intuitively, the reason is that, otherwise, there may be some failure patterns beginning on the prefix and ending on the suffix, that would not be captured (? verdict).

The second objective measures the robustness of the formula (normalized in the $[0, 1]$ interval) by means of bSTL quantitative semantics. As a preliminary step, at the beginning of the execution of the genetic algorithm, every signal in $\mathcal{X}$ is normalized in the $[0, 1]$ interval so that ρ ranges between -1 and 1. This step is handled implicitly and it does not alter the constant value c of constraints $x_i \geq c$ in the generated output formula, which are still represented with their raw, non-normalized value.

This second objective is defined as follows:

$$Rob(\mathcal{X},\varphi) = \frac{2 \cdot |\mathcal{X}| - \sum_{x\in\mathcal{X}} \max_{0\leq i\leq w-1}\{\rho(\varphi,x,i)\} + \sum_{x\in\mathcal{X}} \min_{w\leq i\leq len(x)-1-H(\varphi)}\{\rho(\varphi,x,i)\}}{4 \cdot |\mathcal{X}|}.$$

Since two objectives are taken into consideration, no single best-performing solution can be directly selected from a given population by means of the fitness function. Rather, a *Pareto front* of optimal solutions can be identified, containing all non-dominated solutions.[2]

As a final note, observe that including the window length w in each individual allows each bSTL formula to define its own (optimal) way of splitting the traces: we may indeed expect different kinds of failure, captured by different formulas, to be characterized by different temporal extensions.

D. CROSSOVER

Given two parent solutions, the crossover operation generates two new individuals. As for their computation trees, they are generated by one-point crossover (DEAP's *cxOnePoint*). The operator randomly chooses a node in each individual and exchanges the subtrees rooted at it. To avoid bloat, that is, an excessive increase in mean program size without a corresponding improvement in fitness, we placed a static limit of 17 on the children's height (DEAP's *staticLimit*), following once more a suggestion from Koza [40]. When an invalid (over the height limit) child is generated, it is simply replaced by one of its parents, randomly selected. As for the associated window lengths, they are randomly chosen from the parents. Observe that, in performing the crossover operation, non-valid individuals can be generated concerning the relationship between their horizon and normal behaviour window w; given an individual with formula φ, if $H(\varphi) \geq len(x) - w$, we replace it by one of the parents, randomly chosen.

E. MUTATION

As for the mutation operation, two operators have been used among those available in DEAP, each one chosen with uniform probability: *mutNodeReplacement*, that replaces a randomly chosen node in the individual, and *mutEphemeral*, that changes the value of a single constant used within an individual (including, possibly, the window length). As we did for crossover, in order to control bloat we impose a *staticLimit* constraint equal to 17 to the height of the tree. Moreover, it must be checked whether the resulting individual is valid, with reference to its horizon and window length. If this is not the case, the original individual is returned.

[2] A set $\mathcal{S}$ of solutions for an n-objective problem with fitness function $f = \langle f_1, \ldots, f_n \rangle$ is said to be *non-dominated* if and only if for each $x \in \mathcal{S}$, there exists no $y \in \mathcal{S}$ such that (i) $f_i(y)$ improves $f_i(x)$ for some i, with $1 \leq i \leq n$, and (ii) for all j, with $1 \leq j \leq n$ and $j \neq i$, $f_j(x)$ does not improve $f_j(y)$.

F. SELECTION

To promote population diversity, we rely on the elitist selection strategy implemented in NSGA-III [41], based on the concepts of *reference points* and *niche preservation* (we refer the reader to [41] for details).

G. TERMINATION CRITERIA AND EXTRACTION OF FINAL SOLUTIONS

Let us now focus on the termination criteria of the algorithm and on the extraction of the final solution. As it is commonly done, we impose an upper bound on the number of generations. In addition, we define an *early stopping* strategy, based on the *hypervolume* measure. According to it, the execution of the algorithm is interrupted when no improvement over the hypervolume is observed for a given number of generations. Intuitively, the hypervolume of a Pareto front measures the volume of the search space, bounded by a given reference point, that is weakly dominated by the points on the Pareto front [42]. The assumption is that populations of heterogeneous and well-performing solutions are characterized by a high hypervolume.

Since the EA provides a Pareto front of optimal individuals (φ, w) as its result, to determine the final solution to output we first filter the last population's front keeping all individuals whose formula φ has an accuracy greater than 0.5, that is, better than a random classifier. Then, among such individuals, we return the formula φ of the individual with the highest hypervolume. If no formula with accuracy greater than 0.5 is present in the final front, we return `null`.

H. OTHER HYPERPARAMETERS

The other hyperparameters used by the EA have been established a-priori through grid search tuning performed over a specifically developed synthetic dataset of binary labelled bSTL traces, with the two classes characterized by a heterogeneous set of formulas. They are as follows: *population size* = 100 (tested values $[50, 100, 500, 1000]$); *crossover probability* = 0.7 (tested values $[0.5, 0.6, 0.7, 0.8]$); *mutation probability* = $0.5/\sqrt[2]{num\_gen}$ (tested values $[0.3, 0.4, 0.5, 0.6]$); *max generations* = 500 (a rather conservative upper bound); *hypervolume early stopping* = 25 generations (tested values $[10, 25, 50]$). Note that mutation probability starts rather high to ensure an effective *exploration* of the search space; then, it rapidly decays with the number of generations to foster the *exploitation* of the most promising solutions that have been found. Although we recognize that, in principle, each dataset has a different and optimal set of hyperparameters, as we will see, the above values still provide a solid basis when it comes to the overall framework performance, and can thus be considered default choices.

Another hyperparameter, used by the EA in this specific implementation, based on bSTL and `rtamt`, is *max horizon*, whose meaning is fairly natural. Intuitively, it sets an upper bound on the horizon of the formulas that can be explored within the EA. Enforcing a max horizon h has three effects: first, formulas can capture phenomena that are temporally extended at most $h + 1$ time points (in terms of the sampling rate of the considered time series); second, given the way `rtamt` (equivalently, $eb\text{-}mon$) works, at run time, when evaluating the truth of a formula, the verdict will be ? for the first h time points; third, it has been experimentally observed that the execution time of `rtamt` grows more than linearly with the size of the horizon of a formula. As we will see in Section VI, multiple runs of the framework have been taken into consideration in order to collect statistically relevant data. Thus, to allow for faster experimentation, we set a small value of 20 for *max horizon* on all datasets. Although this might seem restrictive, it still allows us to extract meaningful and well-performing properties. In a general usage scenario, *max horizon* should be set by domain experts considering the previously mentioned three aspects. The impact of the *horizon* length on the framework performance is studied in Section VI-C.

V. THE GENERAL FRAMEWORK

In the following, we describe the proposed framework for preemptive failure detection. As already pointed out, it works in an *online* fashion and it uses the `rtamt` monitoring algorithm to check the incoming system trace for undesired behaviours. As we will see, in terms of binary classification, the occurrence of a bad behaviour is considered as a *positive* event. Thus, a false positive corresponds to an erroneous indication of a bad situation, while a false negative corresponds to a missed detection. Bad behaviours are encoded by bSTL formulas, which are collected in a monitoring pool $\mathcal{P}$.

Operationally, we distinguish between two distinct execution phases of the framework: an optional *warmup* phase and a *runtime* phase. In the first one, the pool $\mathcal{P}$ is populated with an initial set of formulas encoding bad behaviours, following a *teacher forcing*-like approach [43] on supervised training data. In the second one, the framework online monitors the system, starting with a non-empty pool $\mathcal{P}$.

During both phases, $\mathcal{P}$ is iteratively refined by (i) adding new formulas which are able to predict bad behaviours earlier and with increased reliability and coverage, and (ii) removing formulas that are ill-behaved or redundant. In addition to this refinement process autonomously operated by the framework, at any time, domain experts can, in principle, make changes to the pool $\mathcal{P}$, e.g., by manually specifying a new formula encoding a bad behavior.[3]

A. WARMUP EXECUTION PHASE

In the *warmup* phase, we assume that a supervised learning training dataset $\mathcal{X}$ is available, consisting of pairs (x, l), where x represents a system execution trace of length $len(x)$, and l is its corresponding label ($\top$, if x is a trace ending with a failure; $\bot$ otherwise). The overall idea is to monitor, one after the other, all available system traces and, for each of them, to simulate its point-by-point arrival.

[3]It is worth noticing that this manual specification can complement the *warmup* phase when limited supervised training data regarding the system are available.

Algorithm 1: Framework execution (*warmup* phase)

input: initial pool $\mathcal{P}$ of formulas, training dataset $\mathcal{X}$

1: **for** $(x, l) \in \mathcal{X}$ **do**
2: $has\_triggered \leftarrow \bot$
3: $\mathcal{S} \leftarrow \emptyset$
4: **for** $i \leftarrow 0$ **to** $len(x) - 1$ **do**
5: $y \leftarrow x[0, i]$
6: $\mathcal{F} \leftarrow \{\psi \in \mathcal{P} \setminus \mathcal{S} \mid eb\text{-}mon(y, \psi)$ `returns` $\top\}$
7: **if** $\mathcal{F} \neq \emptyset$ **then**
8: UPDATEPOOLINFORMATION($\mathcal{P} \setminus \mathcal{S}, \mathcal{F}, y$)
9: **if** l **then**
10: $has\_triggered \leftarrow \top$
11: $\mathcal{Y} \leftarrow$ GENERATETRAINDATA(y)
12: $\phi \leftarrow$ EXTRACTDISCRFORMULA($\mathcal{Y}$)
13: **if** $\phi \neq$ `null` **then**
14: $far_\phi \leftarrow 0$
15: $\mathcal{P} \leftarrow \mathcal{P} \cup \{\phi\}$
16: **end if**
17: **break**
18: **else**
19: $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{F}$
20: **end if**
21: **end if**
22: **end for**
23: **if** l **and not** $has\_triggered$ **then**
24: $\mathcal{Y} \leftarrow$ GENERATETRAINDATA(x)
25: $\phi \leftarrow$ EXTRACTDISCRFORMULA($\mathcal{Y}$)
26: **if** $\phi \neq$ `null` **then**
27: $far_\phi \leftarrow 0$
28: $\mathcal{P} \leftarrow \mathcal{P} \cup \{\phi\}$
29: **end if**
30: **end if**
31: **end for**
32: **return** $\mathcal{P}$

Algorithm 2: UPDATEPOOLINFORMATION

input: pool $\mathcal{P}$ of formulas, set $\mathcal{F}$ of failure formulas, trace x

1: **for** $\phi \in \mathcal{F}$ **do**
2: $far_\phi \leftarrow (1 - \alpha) \cdot$ NEWFAR(ϕ, x) + $\alpha \cdot far_\phi$
3: **if** $far_\phi > far_{thr}$ **then**
4: REMOVE($\phi, \mathcal{P}$)
5: **end if**
6: **end for**
7: HANDLEREDUNDANCY($\mathcal{P}$)

The *warmup* phase is dealt with by Algorithm 1. The procedure gets, as input, a pool $\mathcal{P}$ of bSTL formulas and the set $\mathcal{X}$ of training system traces. $\mathcal{P}$ may possibly be empty. This is the case when no formula is inserted into it by domain experts. For each training trace x, with label l, two variables are set: *has_triggered*, which keeps track of whether the framework has correctly identified the trace x as a failure one (when $l = \top$); and a set $\mathcal{S}$ of *suspended* formulas. $\mathcal{S}$ includes all formulas that, at some point, erroneously signalled a bad behaviour in x (when $l = \bot$), and are thus ignored by the framework for its operation on the remainder of trace x.

Next, the framework starts the iterative part of its execution, during which the trace x is monitored sequentially, point-by-point. At each iteration i, with $0 \leq i \leq len(x)-1$, the system restricts its attention to the prefix $y = x[0, i]$ of trace x, and it computes the set $\mathcal{F}$ of formulas leading to a *violation* (Algorithm 1, line 6). To this end, it executes the monitoring algorithm `rtamt` that verifies each (non-suspended) formula in $\mathcal{P} \setminus \mathcal{S}$ against the current trace y. Since all formulas are meant to encode bad behaviours, we say that a formula ψ leads to a violation if $eb\text{-}mon(y, \psi)$ returns $\top$ ($eb\text{-}mon$ is defined in Section III).

If at least one violation is detected, procedure UPDATEPOOLINFORMATION is executed (Algorithm 2) to detect and remove redundant or non-reliable formulas from the pool, the latter being formulas issuing several false positives. The procedure will be described in detail later.

Then, if x is an actual failure trace, $\mathcal{P}$ is updated (Algorithm 1, lines 10–17) as follows. Training data to be used for the extraction of a new formula are generated by the function GENERATETRAINDATA (Algorithm 1, line 11). The latter perturbs the execution trace y by adding random Gaussian noise as a counter-overfitting measure, thus producing a set of augmented traces $\mathcal{Y}$ of size n_{aug} (global parameter of the system). Next, function EXTRACTDISCRFORMULA (Algorithm 1, line 12) extracts a (bSTL) formula ϕ that discriminates between *normal* and *failure* (sub)traces obtained from those in $\mathcal{Y}$, by exploiting the evolutionary algorithm as described in Section IV. Notice that ϕ may be `null`, an event that, according to the proposed definition of EA, occurs if none of the formulas in the final front has an accuracy greater than 0.5. If ϕ is not `null`, the initial false alarm rate (FAR) of ϕ is set to 0, and the formula is added to $\mathcal{P}$ (Algorithm 1, lines 13–16).[4] At this point, since the trace x was recognized as a failure one by the framework, the execution on x is halted (Algorithm 1, line 17), and the framework is applied to the next trace in $\mathcal{X}$.

On the contrary, if the framework detected a violation and trace x was not a failure one, all involved formulas $f \in \mathcal{F}$ are *suspended*, meaning that they are not going to be considered by the framework for its execution on the remainder of trace x (Algorithm 1, line 19). This prevents them from repeatedly triggering the extraction of other ill-behaved formulas. Note how formulas in $\mathcal{F}$ are not immediately removed from $\mathcal{P}$, as such an approach would be too aggressive: their false positive detection might not be a generalized behaviour, but something caused by random characteristics of trace x itself. As we will see, false positive detections are still considered

[4]The False Alarm Rate (FAR) is expressed as the ratio between the number of negative events wrongly categorized as positive (false positives) and the total number of actual negative events (false positives + true negatives). Recall that, in our setting, a positive event represents a failure of the monitored system. Formulas with a low FAR are to be preferred.

by the procedure UPDATEPOOLINFORMATION for the maintenance of the pool $\mathcal{P}$. Suspended formulas are reactivated when the next training trace is taken into account by the framework.

The iterative phase of the framework on trace x ends when either x is correctly recognized as a failure trace by a formula in $\mathcal{P}$, or x has run out of points without any failure detection. In the latter case, if trace x was a failure one, we force the formula extraction process (Algorithm 1, lines 23–30). As the last operation of the framework (Algorithm 1, line 32), after being run on every training system trace, the obtained monitoring pool $\mathcal{P}$ is returned.

We would like to conclude this account of the operation of Algorithm 1 by observing that the *warmup* mode draws inspiration from the *teacher forcing* technique employed in deep learning [43]. Such an approach is used here to correct both false positive (Algorithm 1, line 9) and false negative (Algorithm 1, line 19 and line 23) framework predictions. As an example, as we already pointed out, the framework starts its execution with a possibly empty pool $\mathcal{P}$ of properties. Thus, in the most extreme case ($\mathcal{P} = \emptyset$), it cannot identify any bad behaviour of the system. In such a case, the failure is "detected" by observing the training label associated with the training execution trace, an event that forcedly triggers the pool update process. Intuitively, the whole scenario can be thought of as having an oracle assisting and instructing the framework. It is to be expected that, over time, the pool $\mathcal{P}$ becomes large enough so as to allow for the effective detection of bad behaviours of the system, progressively substituting the oracle in its role of correcting false positive and false negative predictions.

Let us focus now on the procedure UPDATEPOOLINFORMATION($\mathcal{P}, \mathcal{F}, x$) (Algorithm 2). Operationally, for each formula $\phi \in \mathcal{F}$ that leads to a violation, the corresponding FAR $far_\phi \in [0, 1]$ is updated. Formulas whose FAR crosses a given threshold far_{thr} (a global parameter of the framework) are removed from the monitoring pool (Algorithm 2, lines 3–4). As already pointed out, a FAR equal to 0 is associated with every formula when added to $\mathcal{P}$. Then, the value of FAR is suitably updated according to the exponential moving average with smoothing constant α, which takes into account the "historical" FAR and the new FAR of the formula (Algorithm 2, line 2); the latter is considered to be 0 if the triggered formula actually anticipated a failure, 1 otherwise (false positive case). In the absence of detailed historical data, assigning an initial FAR equal to 0 to the formulas in the pool is a sensible choice, as they are either defined by a domain expert or generated by the evolutionary algorithm, which in turn optimizes accuracy and robustness measures.

The choice of relying on FAR for the pool maintenance instead of on other "symmetric" performance metrics, say F1-score, is twofold. First, formulas providing several false detections may cause a degradation of the monitoring pool, where other ill-founded formulas are added as a result of their triggering. Thus, they should be avoided at all costs. On the contrary, formulas leading to false negatives do not bring any adverse effect on the monitoring pool, except for increasing its size. The second reason pertains to the very nature of F1-score and similar metrics. To calculate it, it is necessary to establish when a formula experiences both false positives and false negatives. False positives, that is, false detections of bad behaviours, can be easily recognized: if a formula triggers and the forecasted bad event does not occur, that can be unequivocally considered as a false positive. The detection of false negatives is, instead, more subtle, and not well-defined. Indeed, it is perfectly admissible for the system to encounter a total failure not anticipated by any formula in the pool, since they may correctly model completely different failure scenarios. In that case, formulas should not be penalized for the missed detection.

Finally, procedure HANDLEREDUNDANCY($\mathcal{P}$) (Algorithm 2, line 7) removes redundant formulas, i.e., it detects groups of formulas with similar behaviour and keeps a single representative for the entire group (the formula with the lowest FAR or the newest one, if the FAR is the same). To detect the similarity of two formulas, we rely on the Jaccard/Tanimoto test [44] that compares the histories of failures flagged by the formulas along the framework execution.

As a last remark, note how procedure UPDATEPOOLINFORMATION, in the way it is used by Algorithm 1, allows us to continuously update the monitoring pool $\mathcal{P}$ as the training instances are processed, ensuring its quality.

Algorithm 3: Framework execution (*runtime* phase)

input: initial non-empty pool $\mathcal{P}$ of formulas, non-empty set $\mathcal{G}$ of good behaviour training traces, incoming system trace x

1: **while** *true* **do**
2: $\mathcal{F} \leftarrow \{\psi \in \mathcal{P} \mid \textit{eb-mon}(x, \psi)$ `returns` $\top\}$
3: **if** $\mathcal{F} \neq \emptyset$ **then**
4: HANDLEREDUNDANCY($\mathcal{P}$)
5: $\mathcal{X} \leftarrow$ GENERATETRAINDATA(x)
6: $\phi \leftarrow$ EXTRACTDISCRFORMULA($\mathcal{X}$)
7: **if** $\phi \neq$ `null` **then**
8: $far_\phi \leftarrow \text{FAR}(\mathcal{G}, \phi)$
9: **if** $far_\phi \leq far_{thr}$ **then**
10: $\mathcal{P} \leftarrow \mathcal{P} \cup \{\phi\}$
11: **end if**
12: **end if**
13: **end if**
14: **end while**

B. RUNTIME EXECUTION PHASE

Let us now concentrate on the *runtime* phase of the framework which is implemented by Algorithm 3. Here, the framework is used to continuously monitor an incoming trace x, generated by a system during its execution. Other than the trace, the procedure gets in input a pool $\mathcal{P}$ of properties, that can be assumed to be non-empty, either because it is

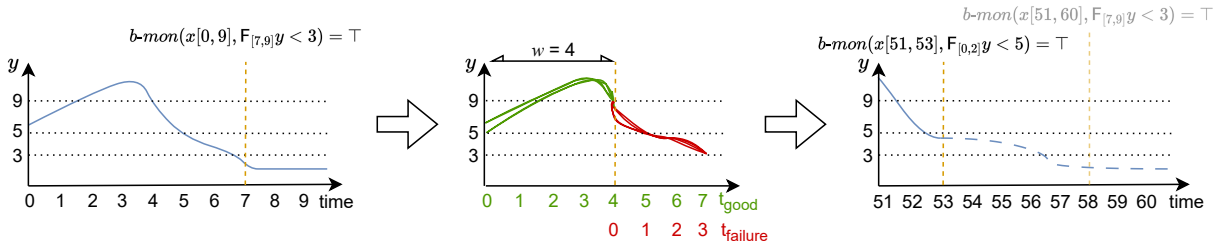


FIGURE 2. A simplified example of the framework operation. Good and failure behaviour subtraces are represented in green and red, respectively.

returned by Algorithm 1 or hand-filled by domain experts. In addition, it takes into consideration a non-empty set $\mathcal{G}$ of past good execution traces of the system. The latter can be either extracted from the training warmup data, if available, or derived directly from the execution history of the system, restricting to those portions that are sufficiently far from failure events, following the suggestions of domain experts.

Algorithm 3 behaves as follows. At each time step, the set $\mathcal{F}$ of formulas leading to a *violation* is computed (Algorithm 3, line 2) by executing the monitoring algorithm `rtamt`, which checks each formula in $\mathcal{P}$ against the incoming system trace x. If at least one formula is triggered, $\mathcal{P}$ is updated (Algorithm 3, lines 3–13). First, procedure HANDLEREDUNDANCY is called, to identify and remove from the pool $\mathcal{P}$ possible redundant formulas, exactly in the same way as in the warmup phase. Next, training data to be used for the extraction of a new formula are generated by the function GENERATETRAINDATA(x). Finally, function EXTRACTDISCRFORMULA($\mathcal{X}$) extracts a (bSTL) formula ϕ that discriminates between *normal* and *failure* (sub)traces by using the EA. If the formula ϕ generated by the EA is not `null`, its FAR is computed with respect to the reference set of good traces $\mathcal{G}$ and, if such a value is less than or equal to the threshold far_{thr}, the formula is added to the pool $\mathcal{P}$.

As it can be noticed, the main differences between the warmup and runtime phases are that, in the latter, there is no teacher forcing, and thus the entire failure detection task is carried out by means of monitoring; moreover, the FAR of a formula is established only once by considering all traces in the reference set $\mathcal{G}$, being the latter fixed.

As a final remark, note that Algorithm 3 (runtime) can, in principle, be run independently from Algorithm 1 (warmup), if there is at least one property in $\mathcal{P}$ and a set of good traces $\mathcal{G}$ of the considered system is available. This allows one to use the framework in a runtime setting even in the absence of supervised training data, as long as at least one failure property has been provided by domain experts and some (portions of) unlabeled past execution traces of the system, that express good/normal behaviours, are accessible.

An intuitive account of the operation of the framework is depicted in Figure 2. The framework is first attached to a trace x generated by the system for its runtime monitoring, with a pool $\mathcal{P}$ containing just the formula $\phi = \mathsf{F}_{[7,9]}y < 3$ (left picture). Function $eb\text{-}mon(x, \phi)$ is run against the incoming trace and, specifically, $b\text{-}mon(x[0, 9], \phi)$ identifies a failure occurring at time point 7. This leads to the extraction of a new formula. To this end, trace $x[0, 7]$ is augmented, and then the EA is run on the set of resulting traces. In the middle picture, just for illustrative purposes, an exemplary splitting of the augmented traces based on a window length $w = 4$ is reported. Each trace is partitioned into a good behaviour prefix and a failure suffix. For formula evaluation purposes, in the EA each subtrace is considered as to be starting from index 0. As a result, the formula $\psi = \mathsf{F}_{[0,2]}y < 5$, which is able to distinguish between the augmented prefixes and suffixes, is generated and added to the pool $\mathcal{P}$. Finally, a subsequent part of the operation of the framework is described (right picture) Here, the recently discovered formula ψ identifies a failure occurring at time point 53 ($b\text{-}mon(x[51, 53], \psi) = \top$). Without such a formula, ϕ would have detected a violation only with respect to time point 58.

For the sake of convenience, all the global parameters of the framework are listed in Table 1, with an intuitive account and a short description of their expected behaviour.

VI. EXPERIMENTAL EVALUATION

In this section, we give a detailed account of the experimental evaluation of the framework on 3 public datasets. In addition, we make a comparison with previous results from the literature. First, we introduce the datasets; then, we describe the experimental workflow; finally, the obtained results are portrayed. We pay particular attention to interpretability issues.

A. DATASETS

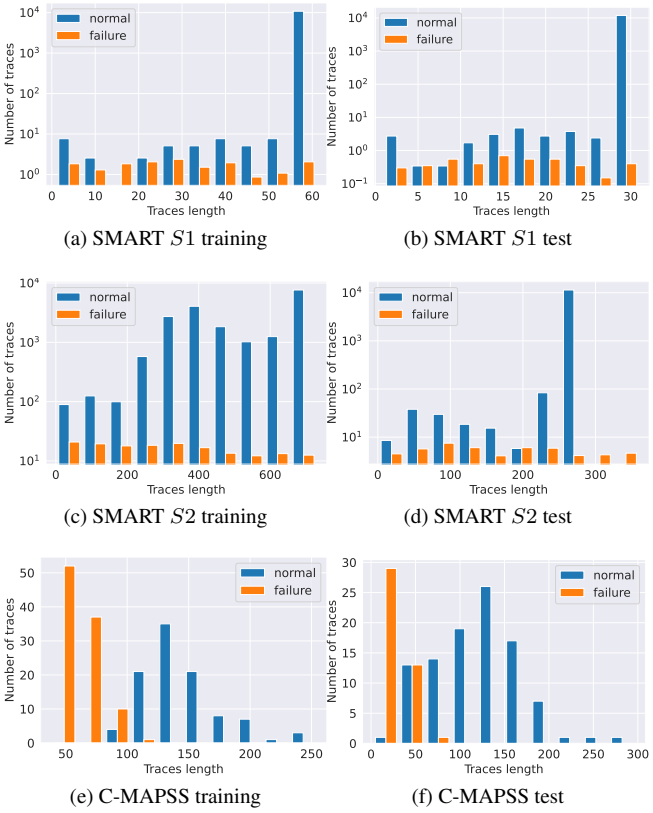
We considered the datasets Backblaze Hard Drive[5], Tennessee Eastman Process[6], and NASA C-MAPSS[7].

The *Backblaze Hard Drive* dataset (also referred to as *SMART* dataset hereafter) contains continuously updated information on the "health" status of hard drives in the Backblaze data centre. Here, we focus on Self Monitoring Analysis and Reporting Technology (SMART) attributes of the ST4000DM000 hard drive model recorded daily from

[5]https://www.backblaze.com/b2/hard-drive-test-data.html
[6]https://doi.org/10.7910/DVN/6C3JR1
[7]https://data.nasa.gov/dataset/C-MAPSS-Aircraft-Engine-Simulator-Data/xaut-bemq

TABLE 1. Global parameters of the framework.

	Description	Value	Search range	Expected behaviour
α	*smoothing constant* for formulas FAR update	0.9	$\{0.7, 0.8, 0.9\}$	A lower value gives greater weight to new scores. Such a behaviour is preferred, for instance, in the event of a system undergoing rapid changes.
far_{thr}	*maximum allowed FAR* for formulas in the pool	0.2	$\{0.1, 0.2, 0.3\}$	A low value leads to more reliable formulas being kept in the monitoring pool. Still, in the presence of noise/outliers in the monitored traces, it may cause a formula to be inadvertently removed from the pool. There, a higher value should be considered.
n_{aug}	*number of augmentations* for each failure trace	100	$\{50, 100, 150\}$	A large value should help avoid overfitting; however, it also increases the computational burden of the formula extraction phase.



(a) SMART $S1$ training (b) SMART $S1$ test

(c) SMART $S2$ training (d) SMART $S2$ test

(e) C-MAPSS training (f) C-MAPSS test

FIGURE 3. Trace length distributions.

2015 to 2017. Each trace is described by the following features: the date of the report, the serial number of the drive, a label indicating a drive failure and 21 SMART parameters with both discrete and real values. To compare the framework with the literature, two training/test set splits are considered:

- Split $S1$: training set from October to November 2016 with 0.68% failure traces, test set December 2016 with 0.64% failure traces. The total number of traces is 34970, and their length distributions are depicted in Figure 3;
- Split $S2$: training set from January 2015 to December 2016 with 0.71% failure traces, test set January to December 2017 with 0.66% failure traces. The total number of traces is 36242, and their length distributions are depicted in Figure 3;

The *Tennessee Eastman Process* (TEP) dataset contains simulated data from a fictitious chemical plant. This dataset includes 1000 training and 1000 test traces labelled with Type 0 (normal behaviour) or Type 1 (faulty behaviour) sampled every 3 minutes. Each training trace lasts for 25 hours, whereas test traces last for 48 hours. There are 500 faulty traces in both sets. Each simulation is represented by a multivariate time series with the following features: trace ID, fault type, and 52 variables tracking data about the operating values of plant components.

The *NASA Commercial Modular Aero-Propulsion System Simulation* (C-MAPSS) dataset includes run-to-failure simulated data of turbofan jet engines. Specifically, in the considered dataset FD001, engines are simulated according to a single operating 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 sensors. Although each trace 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 trace length distributions are depicted in Figure 3. The dataset includes 100 training traces, each ending with a failure, and 100 test traces, each ending an arbitrary and known number of time steps before the failure (gap). In order to compare our framework with the literature [45], failure traces are generated by considering the 30% suffix of each engine observation as faulty, and the remaining 70% prefix as normal behaviour. Thus, this leads to 200 training traces. On the other hand, 43 failure and 100 normal behaviour traces are generated for the test set. The reason is that, given a test set trace, the 30% suffix is computed over the trace length including the gap and, thus, it may result to be empty.

B. EXPERIMENT SETUP

For each dataset, we performed the initial warmup phase by running Algorithm 1 on a sample of training execution traces related to both malfunctions (failure traces) and good executions. The traces were considered by the framework one after the other, according to a random ordering.

Once the warmup phase ended, the framework was evaluated on test set traces (Algorithm 3) in two modes: the *online* mode, where the framework continues to learn new properties from the execution traces, and the *offline* mode, where the

TABLE 2. Experimental results.

Dataset	Approach	Precision	Recall	FAR	F1-score
SMART S1	[9]	0.87	0.41	0.00	0.55
	[46]	0.51	0.54	0.00	0.52
	our	0.54	0.60	0.00	0.56
SMART S2	[9]	0.98	0.87	0.01	0.92
	[47]	0.91	0.94	0.05	0.93
	our	0.89	0.97	0.00	0.93
TEP	[48]	1.00	1.00	–	1.00
	[49]	1.00	1.00	0.00	1.00
	our	1.00	1.00	0.00	1.00
C-MAPSS	[45]	0.71	1.00	–	0.83
	our	0.96	0.77	0.01	0.86

Note: the results of [46], [47], [48], [49], and [45] are listed as reported in the original references; those of [9] have been determined by the authors of this work running the code made available in the original publication on the considered datasets. Numbers are rounded to two decimal places.

properties in the monitoring pool are not updated, so that only the properties learnt in the warmup phase are taken into account when predicting failures on test set traces. This latter mode was useful to compare the proposed solution with those from the literature, while the former let us determine how the values of the considered metrics evolved over time. The two test set evaluation modes were carried out on a random ordering of both good and failure traces.

The performance of the two test phases was evaluated in terms of *precision*, *recall*, *FAR*, and *F1-score*. Let TP be the number of true positives, that is, bad behaviours identified as such, FP be the number of false positives, TN be the number of true negatives, and FN be the number of false negatives. The metrics are defined as follows:

$$precision = \frac{TP}{TP + FP} ,$$

$$recall = \frac{TP}{TP + FN} ,$$

$$FAR = \frac{FP}{FP + TN} ,$$

$$F1\text{-}score = \frac{2 \cdot precision \cdot recall}{precision + recall} .$$

All experiments were run 10 times varying the random seed governing the order in which execution traces are presented to the framework during the warmup and the online test phases, so as to collect statistical data regarding the considered metrics.

C. RESULTS

To begin with, the global parameters of the framework, chosen through grid search optimization on training set data, are shown in Table 1. In the remainder of the section, we will assess the framework performance in several respects. First, we will present the results of the *offline* and the *online* evaluation, as described in Section VI-B. Then, we will focus on the impact of teacher forcing and formula *max horizon*.

Offline evaluation

As for the *offline evaluation* mode, we compared the proposed solution with other state-of-the-art approaches to failure detection on the three previously-described datasets. Given the continuously updating nature of the Backblaze dataset, we focused our analysis on two studies that take into account the specific versions we consider, namely, those reported in [46] and [47]. The first one [46] makes use of a feed-forward neural network model on split $S1$, while the second one [47] evaluates a Long-Short Term Memory (LSTM) recurrent neural network on split $S2$. In addition, we took into account a third proposal [9], that is, a model obtained by combining a convolutional neural network (CNN) and an LSTM recurrent neural network, applying it to both $S1$ and $S2$ splits, following the setup outlined by the authors for the SMART features group. As for the case of fault detection on the TEP dataset, we considered an approach based on image processing techniques along with feed-forward and radial basis function neural networks [48], and a solution based on a nonlinear support vector machine [49]. Finally, as for the C-MAPSS case study, we compared our framework with a solution based on a CNN model presented in [45].

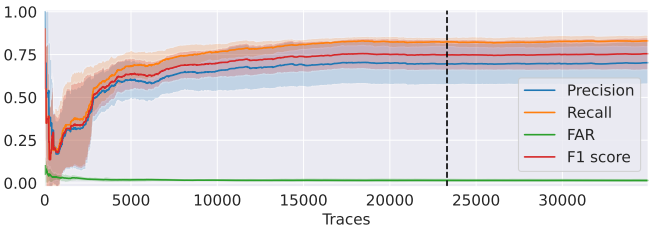
Results achieved by the above solutions are reported in Table 2, together with those of the proposed framework (label *our*). Our solution exhibits an average performance on par with the considered state-of-the-art ones. This is even more relevant if we also bear in mind that all of the contenders provide no explanation for the predicted failures. On the contrary, a distinguishing feature of the proposed approach, compared to previous ones, is that it is interpretable: it relies on the extraction of properties expressed as temporal logic formulas, that provide an understandable explanation of the undesired behaviors of the system. Moreover, they can be subsequently exploited for tasks such as root cause analysis, diagnosis, and preemptive failure detection.

Online evaluation

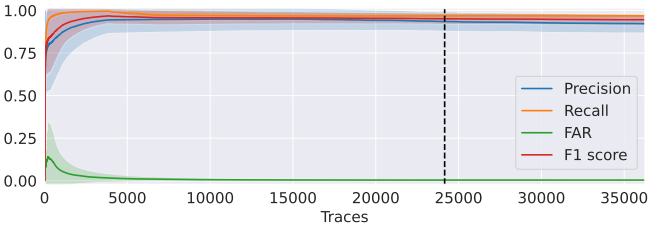
As for the *online evaluation* mode, results for the SMART, TEP, and C-MAPSS datasets are shown in Figure 4. Note that, as the number of traces seen by the framework increases, a slight but consistent improvement of the metrics occurs. This is not obvious, since in such a case maintaining a good performance requires the ability to discover new properties able to reflect the evolution of the behaviour of the monitored system over time.

Figure 5 illustrates, for each considered dataset, the average number of formulas in the monitoring pool at each warmup and runtime iteration. Note that, at certain iterations, there is a decrease in the pool size. This happens when formulas are removed because they are redundant.

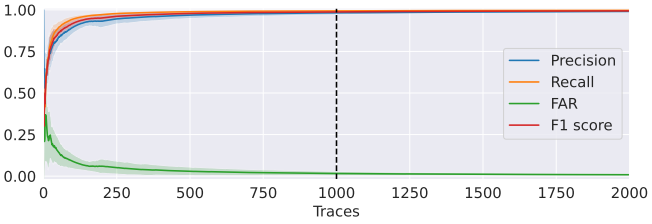
As an example of removal from the pool, we present the case in which two properties, $\mathsf{F}_{[30,45]} SENSOR_{11} < 49.60$ and $\mathsf{F}_{[36,41]} SENSOR_{11} < 49.77 \wedge \mathsf{F}_{[45,52]}$



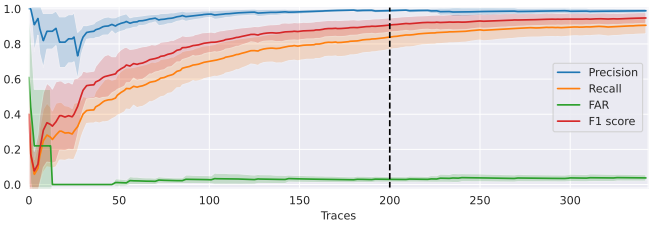
(a) SMART $S1$



(b) SMART $S2$



(c) TEP



(d) C-MAPSS

FIGURE 4. Metrics average and standard deviation for the considered datasets. The vertical dashed line represents the transition from warmup to online traces.

$SENSOR_{11} < 49.39$, were extracted in a framework execution on the C-MAPSS dataset. After a series of iterations, their failure detection histories were, respectively, $[1,0,0,1,0,1,1,0,0,1,0,1,1,0,0,1,1,1]$ and $[1,0,1,0,0,1,1,0,1,0,1,1,1]$, while the FAR values were 0.07 and 0.0. Since both properties were considered to be redundant according to the Jaccard/Tanimoto test, the framework kept only the second one, by reason of its lower FAR. From a domain perspective, the two formulas refer to a similar behavior of the same sensor ($SENSOR_{11}$) which indicates a loss of static pressure in the *high-pressure compressor outlet*.

Let us now consider some examples of the bSTL formulas used within the framework. An example for the Backblaze dataset is formula $(\mathsf{G}_{[0,2]}SMART_{194}{>}45.6) \wedge (\mathsf{F}_{[2,3]}SMART_{198}{>}0.32)$. Such a formula makes evident a bad behavior where the hard drive maintains a temperature exceeding 45.6 °C in the first 3 days, and then, in the following 2 days, its *uncorrectable sector count* becomes greater than 0. As another example of framework execution, consider the formula $f_1 = \mathsf{F}_{[0,19]}SMART_{198} > 2.59$, extracted (and added to the monitoring pool) during an iteration of the framework. According to the definitions of the SMART attributes, sensor $SMART_{198}$ is a critical one and f_1 expresses the fact that the threshold 2.59 of *sector read/write errors* is exceeded. During a later iteration of the *warmup* phase, a failure prediction is issued thanks to the triggering of f_1. As a consequence, $f_2 = \mathsf{F}_{[1,16]}SMART_{189} > 8.28$ is extracted, meaning that a certain number (8.28) of *unsafe fly height conditions* is reached before the critical number of sector read/write errors is exceeded. This pattern is quite reasonable, as it describes a case in which the disk head is operating at an unsafe height, ultimately damaging a disk sector and consequently causing read and write errors. Notice that the framework allows us to predict a failure based on sensor $SMART_{189}$, which is not considered to be critical in the SMART specification, by uncovering a pattern linking it to the critical sensor $SMART_{198}$.

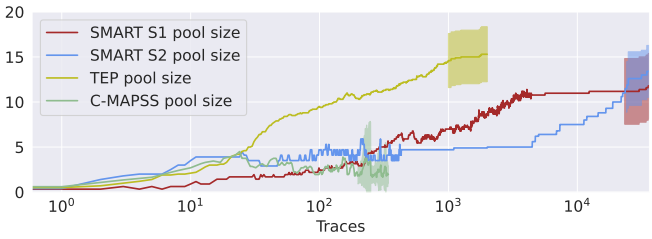


FIGURE 5. Average and standard deviation of the pool size for each framework iteration in the case of the SMART S1, SMART S2, TEP, and C-MAPSS datasets, in both warmup (transparent area) and runtime (opaque area) phases. The x-axis is on a logarithmic scale.

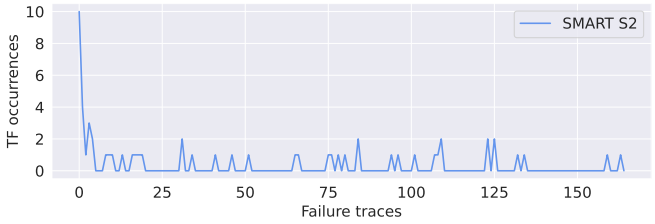
Turning to the TEP dataset, an extracted formula is $(\mathsf{G}_{[1,4]}XMEAS_{21} > 94.6) \wedge (\mathsf{G}_{[2,4]}XMEAS_{20} > 341)$. It reveals a bad behaviour where the *compressor* is operating with a power greater than 341 kW, while the temperature of the *reactor of the plant* exceeds 94.6 °C.

As for the C-MAPSS dataset, the formula $(SENSOR_{10} < 1.3) \wedge (\mathsf{F}_{[4,6]}SENSOR_{11} < 47.62)$ was generated, which signals a bad behaviour where a loss of pressure in the *high-pressure compressor outlet* follows a loss of pressure in the *engine*. Notice that the subformula $SENSOR_{10} < 1.3$ does not contain any time operator, meaning that it is evaluated at the currently observed time point.

Impact of teacher forcing

Figure 6 reports the number of teacher forcing interventions during the warmup phase, as more and more failure traces are encountered. Specifically, for each amount of encountered failure traces, the sum over multiple (10) framework executions is reported. As expected, teacher forcing triggers mainly at the beginning of the warmup phase, when the monitoring pool is empty. As formulas are learned over time, teacher forcing interventions decrease till a stationary

(a) SMART $S1$



(b) SMART $S2$

(c) TEP

(d) C-MAPSS

FIGURE 6. Teacher forcing interventions during the warmup phase. For each amount of encountered failure traces, the sum over multiple (10) framework executions is reported.

behaviour is reached. Of course, the latter depends on the specific dataset, and it confirms what was to be expected from the performance reported in Table 2. As an example, on the TEP dataset, where an F1 score of 1.0 is achieved, the number of teacher forcing interventions rapidly approaches 0.

Impact of max horizon

Figure 7 reports, for a single execution of the framework, the offline mode performance on each dataset, as obtained by varying the *max horizon* value. Once more, different datasets exhibit different behaviours. Although results might appear rather counterintuitive (setting a large *max horizon* does not prevent, in principle, the discovery of formulas with shorter horizons), following a preliminary analysis, they are likely due to an overfitting effect. Indeed, formulas with a larger horizon have the capability of capturing more detailed and extended phenomena, that could be highly trace-dependent.

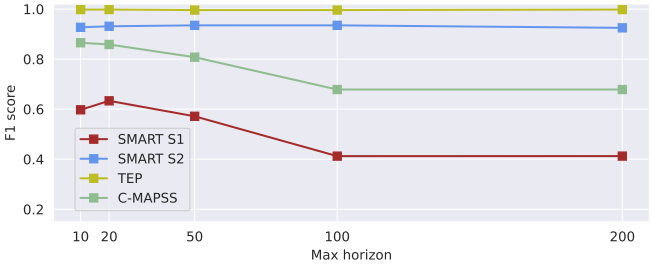


FIGURE 7. Impact of maximum horizon value in test F1 score for different datasets.

Moreover, we would like to recall that the concept of horizon does not apply to the framework in general, but is tied to the particular kind of logic and monitoring tool employed here.

VII. STRENGTHS AND LIMITATIONS

The proposed framework relies on approaches originating from the two fields of machine learning and formal methods, combining their strengths in an effective way. More precisely, the former domain provided us with tools and techniques for the extraction of properties from temporal data, while the latter allowed us to formalize such properties by means of logic formulas and to online monitor a given system against them in a principled manner. The key feature of the proposed approach is its interpretability: as shown in Section VI-C, by means of the extracted logical formulas, the framework gives an understandable account of settings leading to future failures, allowing domain experts to take appropriate action and enriching their overall knowledge. While contributions from the literature show that interpretability is often achieved at the expense of prediction accuracy, e.g., by relying on a simple white-box model instead of a more complex black-box one, quantitative results showed that the performance of the proposed approach is on par with previous, non-interpretable solutions.

As a final note, while in this work we applied the proposed framework to the domain of failure detection, similar ideas can in principle be employed to detect and predict any type of event or anomaly, whether positive or negative in nature. Among the first, we mention a spike in the history of sales of a retail store, or a generalized increase in the grade point average of students enrolled in the latest edition of a course; among the latter scenarios, the detection of seizures in hospitalized patients based on their continuously recorded vital signs, or the identification of violations of a level of service agreement in the context of a contract between a service provider and a customer.

We would like to conclude this section by pointing out some limitations of the framework. First, in the considered datasets, all traces come from the same plant (resp., hard disk, jet engine model) operating under the same conditions. To deal with more than one type of system, separated monitoring pools have to be employed to prevent conflicts among formulas. Second, the considered datasets only deal with numerical

data. It is worth evaluating the proposed approach on datasets encompassing categorical data, naturally leading to the usage of other logics, like, for instance, LTL. Third, Algorithm 3 operates in a sequential fashion: (i) the system is monitored until a formula is satisfied by the incoming data; (ii) such an event triggers the phase of the property extraction, that results in the addition of a new formula to the pool; (iii) then, the monitoring of the system resumes. Although this behaviour is perfectly acceptable for a prototype implementation applied on benchmark datasets, as the one described here, a multithreaded version, able to update the property pool asynchronously, while monitoring the system, remains to be developed. Finally, Algorithm 3 makes use of a fixed reference set of good behaviour traces to prevent formulas with a high FAR to be added to the monitoring pool. This is definitely a reasonable approach, but it does not take into account changes in the behaviour of the monitored system, that may happen due to, for instance, updates, upgrades, or degradation phenomena. To overcome this limitation, we may think of extracting new normal behaviour traces from runtime data and adding them to the reference set.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a novel general framework for runtime system verification that combines monitoring with machine learning, to be used for early failure detection over streams of data. Experimental results showed that it is able to issue failure warnings in an anticipatory and effective manner and to incrementally learn new specifications to be monitored against the considered system.

As for future work, we would like to underline the following directions: (i) the application of the framework to other datasets; (ii) user tests to assess the quality of interpretability (iii) the experimentation with other logics, including an extension of STL dealing with categorical data [50]; and (iv) the development of a multithreaded version of the framework, able to asynchronously deal with the update of the property pool while monitoring the system.

REFERENCES

[1] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[2] 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- and Post-Deployment Verification Techniques (PrePost@iFM)*. Springer, 2017, pp. 15–28.

[3] M. Gerhold, A. Hartmanns, and M. Stoelinga, "Model-based testing of stochastically timed systems," *Innovations in Systems and Software Engineering*, vol. 15, no. 3-4, pp. 207–233, 2019.

[4] P. Korvesis, S. Besseau, and M. Vazirgiannis, "Predictive maintenance in aviation: Failure prediction from post-flight reports," in *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 1414–1422.

[5] S. Vallachira, M. Orkisz, M. Norrlöf, and S. Butail, "Data-driven gearbox failure detection in industrial robots," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 1, pp. 193–201, 2019.

[6] B. Mohammed, I. Awan, H. Ugail, and M. Younas, "Failure prediction using machine learning in a virtualised HPC system and application," *Cluster Computing*, vol. 22, no. 2, pp. 471–485, 2019.

[7] J. Gao, H. Wang, and H. Shen, "Task failure prediction in cloud data centers using deep learning," *IEEE Transactions on Services Computing*, vol. 15, pp. 1411–1422, 2022.

[8] K. Aggarwal, O. Atan, A. K. Farahat, C. Zhang, K. Ristovski, and C. Gupta, "Two birds with one network: Unifying failure event prediction and time-to-failure modeling," in *Proceedings of the 6th IEEE International Conference on Big Data (BigData)*. IEEE, 2018, pp. 1308–1317.

[9] S. Lu, B. Luo, T. Patel, Y. Yao, D. Tiwari, and W. Shi, "Making disk failure predictions SMARTer!" in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 2020, pp. 151–167.

[10] G. Petmezas, K. Haris, L. Stefanopoulos, V. Kilintzis, A. Tzavelis, J. A. Rogers, A. K. Katsaggelos, and N. Maglaveras, "Automated atrial fibrillation detection using a hybrid CNN-LSTM network on imbalanced ECG datasets," *Biomedical Signal Processing and Control*, vol. 63, p. 102194, 2021.

[11] 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 (IJCAI 2019)*. IJCAI Organization, 2019, pp. 5591–5598.

[12] D. Neider and I. Gavran, "Learning linear temporal properties," in *Proceedings of the 18th Formal Methods in Computer Aided Design (FMCAD 2018)*. IEEE, 2018, pp. 1–10.

[13] G. Bombara, C. I. Vasile, F. Penedo, H. Yasuoka, and C. Belta, "A decision tree approach to data classification using Signal Temporal Logic," in *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control (HSCC 2016)*. ACM, 2016, pp. 1–10.

[14] S. Mohammadinejad, J. V. Deshmukh, A. G. Puranic, M. Vazquez-Chanlatte, and A. Donzé, "Interpretable classification of time-series data using efficient enumerative techniques," in *Proceedings of the 23rd ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2020)*. ACM, 2020, pp. 1–10.

[15] L. Nenzi, S. Silvetti, E. Bartocci, and L. Bortolussi, "A robust genetic algorithm for learning temporal specifications from data," in *Proceedings of the 15th International Conference on Quantitative Evaluation of Systems (QEST)*, vol. 11024. Springer, 2018, pp. 323–338.

[16] G. Bombara and C. Belta, "Offline and online learning of signal temporal logic formulae using decision trees," *ACM Transactions on Cyber-Physical Systems*, vol. 5, no. 3, pp. 1–23, 2021.

[17] G. Chen, M. Liu, and Z. Kong, "Temporal-logic-based semantic fault diagnosis with time-series data from industrial internet of things," *IEEE Transactions on Industrial Electronics*, vol. 68, no. 5, pp. 4393–4403, 2020.

[18] C. Da Costa, M. H. Mathias, P. Ramos, and P. S. Girão, "A new approach for real time fault diagnosis in induction motors based on vibration measurement," in *Proceedings of the 27th IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*. IEEE, 2010, pp. 1164–1168.

[19] C. Dousson and T. V. Duong, "Discovering chronicles with numerical time constraints from alarm logs for monitoring dynamic systems," in *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, 1999, pp. 620–626.

[20] M. Kashiwagi, C. da Costa, and M. Mathias, "Development of diagnosis system for rolling bearings faults on real time based on FPGA," *Renewable Energy and Power Quality Journal*, vol. 10, no. 10, pp. 545–549, 2012.

[21] P. Naldurg, K. Sen, and P. Thati, "A temporal logic based framework for intrusion detection," in *Proceedings of the 24th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, vol. 3235. Springer, 2004, pp. 359–376.

[22] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, K. Niekum, and U. Topcu, "Safe reinforcement learning via shielding," in *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, (AAAI)*. AAAI Press, 2018, pp. 2669–2678.

[23] S. Bufo, E. Bartocci, G. Sanguinetti, M. Borelli, U. Lucangelo, and L. Bortolussi, "Temporal logic based monitoring of assisted ventilation in intensive care patients," in *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, vol. 8803. Springer, 2014, pp. 391–403.

[24] Y. Watanabe, H. Otsuka, M. Sonoda, S. Kikuchi, and Y. Matsumoto, "Online failure prediction in cloud datacenters by real-time message pattern learning," in *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2012)*. IEEE, 2012, pp. 504–511.

[25] D. Kim, S. C. Han, Y. Lin, B. H. Kang, and S. Lee, "RDR-based knowledge based system to the failure detection in industrial cyber physical systems," *Knowledge-Based Systems*, vol. 150, pp. 1–13, 2018.

[26] B. R. Gaines and P. Compton, "Induction of ripple-down rules applied to modeling large databases," *Journal of Intelligent Information Systems*, vol. 5, no. 3, pp. 211–228, 1995.

[27] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL specification mining (T)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 2015, pp. 81–92.

[28] A. J. Shah, P. Kamath, J. A. Shah, and S. Li, "Bayesian inference of temporal task specifications from demonstrations," in *Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS)*. NeurIPS Foundation, 2018, pp. 3808–3817.

[29] D. Kasenberg and M. Scheutz, "Interpretable apprenticeship learning with temporal logic specifications," in *Proceedings of the 56th IEEE Conference on Decision and Control (CDC 2017)*. IEEE, 2017, pp. 4914–4921.

[30] T. D. B. Le and D. Lo, "Deep specification mining," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2018, pp. 106–117.

[31] 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 (OVERLAY@AIxIA 2019)*, vol. 2509. CEUR-WS.org, 2019, pp. 71–76.

[32] A. Brunello, D. Della Monica, A. Montanari, and A. Urgolo, "Learning how to monitor: Pairing monitoring and learning for online system verification," in *Proceedings of the 2nd Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis (OVERLAY@BOSK 2020)*, vol. 2785. CEUR-WS.org, 2020, pp. 83–88.

[33] E. M. Clarke, O. Grumberg, D. Kroening, D. A. Peled, and H. Veith, *Model checking*, 2nd ed. MIT Press, 2018.

[34] Z. Manna and A. Pnueli, *Temporal verification of reactive systems - safety*. Springer, 1995.

[35] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, 2004, pp. 152–166.

[36] D. Ničković and T. Yamaguchi, "RTAMT: Online robustness monitors from STL," in *Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, vol. 12302. Springer, 2020, pp. 564–571.

[37] A. E. Eiben and J. E. Smith, *Introduction to evolutionary computing*. Springer, 2003.

[38] R. Poli, W. Langdon, and N. Mcphee, *A Field Guide to Genetic Programming*. http://www.gp-field-guide.org.uk, 2008.

[39] F. A. Fortin, F. M. De Rainville, M. A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, no. 70, pp. 2171–2175, 2012.

[40] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Statistics and Computing*, vol. 4, no. 2, pp. 87–112, 1994.

[41] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, 2013.

[42] Y. Cao, B. J. Smucker, and T. J. Robinson, "On using the hypervolume indicator to compare Pareto fronts: Applications to multi-criteria optimal experimental design," *Journal of Statistical Planning and Inference*, vol. 160, pp. 60–74, 2015.

[43] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989.

[44] N. C. Chung, B. Miasojedow, M. Startek, and A. Gambin, "Jaccard/-Tanimoto similarity test and estimation methods for biological presence-absence data," *BMC Bioinformatics*, vol. 20, no. 15, pp. 1–11, 2019.

[45] T. S. Kim and S. Y. Sohn, "Multitask learning for health condition identification and remaining useful life prediction: Deep convolutional neural network approach," *Journal of Intelligent Manufacturing*, vol. 32, no. 8, pp. 2169–2179, 2020.

[46] X. Huang, "Hard drive failure prediction for large scale storage system," Ph.D. dissertation, University of California, Los Angeles, 2017.

[47] C.-J. Su and Y. Li, "Recurrent neural network based real-time failure detection of storage devices," *Microsystem Technologies*, vol. 28, no. 2, pp. 621–633, 2019.

[48] P. Hajihosseini, M. M. Anzehaee, and B. Behnam, "Fault detection and isolation in the challenging Tennessee Eastman Process by using image processing techniques," *ISA Transactions*, vol. 79, pp. 137–146, 2018.

[49] M. Onel, C. A. Kieslich, and E. N. Pistikopoulos, "A nonlinear support vector machine-based feature selection approach for fault detection and diagnosis: Application to the Tennessee Eastman Process," *American Institute of Chemical Engineers Journal*, vol. 65, no. 3, pp. 992–1005, 2019.

[50] A. Rodionova, E. Bartocci, D. Nickovic, and R. Grosu, "Temporal logic as filtering," in *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control (HSCC)*. ACM, 2016, pp. 11–20.

ANDREA BRUNELLO got his PhD in Computer Science in 2020 from the University of Udine (Italy). He then joined as a post-doc the Department of Mathematics, Computer Science, and Physics at the University of Udine, where he is now an assistant professor. His main research interests are in data modelling and integration, data mining, and machine learning. He has published various papers in international conferences and journals, and he is supervisor of theses.

DARIO DELLA MONICA is associate professor at University of Udine (Italy). He obtained a PhD in Computer Science at University of Udine in 2011. His main interests are in logic for computer science, formal methods, and automatic verification. He has published around 50 contributions in international peer-reviewed journals and conferences.

ANGELO MONTANARI is full professor of Computer Science at the University of Udine (Italy), where he chairs the Data Science & Automatic Verification Lab. He got his PhD in Logic and Computer Science from the University of Amsterdam (The Netherlands) in 1996. His main research interests are in formal methods, artificial intelligence, and spatio-temporal databases. He has published over 250 contributions in international journals, conferences, and handbook chapters.

NICOLA SACCOMANNO is a PhD student at the Department of Mathematics, Computer Science, and Physics of the University of Udine (Italy), where he got a Master Degree in Computer Science in 2019 (jointly with the University of Klagenfurt, Austria). His main research interests are in applied machine learning and artificial intelligence, specifically with applications to indoor positioning and healthcare domains. He has published several works in international conferences, journals, and he co-supervised multiple students.

ANDREA URGOLO received his M.S. degree in Computer Science from the University of Udine (Italy). He is currently pursuing the Ph.D. degree in Computer Science, Mathematics and Physics. His main research interests include Data Science, Machine Learning and Formal Methods applied to the smart sensing and predictive maintenance domains.