Learning What to Monitor: Using Machine Learning to Improve past STL Monitoring

Andrea Brunello, **Luca Geatti**, **Angelo Montanari** and **Nicola Saccomanno**
University of Udine, Udine, Italy
andrea.brunello@uniud.it,luca.geatti@uniud.it,angelo.montanari@uniud.it,nicola.saccomanno@uniud.it

Abstract

Monitoring is a runtime verification technique that can be used to check whether an execution of a system (*trace*) satisfies or not a given set of properties. Compared to other formal verification techniques, e.g., model checking, one needs to specify the properties to be monitored, but a complete model of the system is no longer necessary. First, we introduce the *pure past* fragment of *Signal Temporal Logic* (ppSTL), and we use it to define the monitorable safety (G(ppSTL)) and cosafety (F(ppSTL)) fragments of STL, which properly extend the commonly-used bounded-future fragment. Then, we devise a multi-objective genetic programming algorithm to automatically extend the set of properties to monitor on the basis of the history of failure traces collected over time. The framework resulting from the integration of the monitor and the learning algorithm is then experimentally validated on various public datasets. The outcomes of the experimentation confirm the effectiveness of the proposed solution.

1 Introduction

In this paper, we propose a tight integration of monitoring and machine learning for preemptive failure detection, with formal guarantees on interpretability, monitorability, and expressiveness.

Monitoring is a runtime verification technique for the analysis of complex systems [Leucker and Schallhart, 2009]. It consists of the generation of a *monitor* that is paired with the system under analysis, and it reports a positive (resp., negative) verdict whenever the current execution, and all its continuations, are guaranteed to be good (resp., bad). Therefore, verdicts of monitors are always *irrevocable*. Expressing good and bad behaviors to be monitored is done by means of temporal logics, in particular *Signal Temporal Logic* (STL) which proved to be quite effective in this context [Maler and Nickovic, 2004]. Not all formulas of STL are *monitorable*: there are formulas for which it is impossible to produce a verdict by a finite number of observations [Bauer *et al.*, 2011]. Although not being able to check exhaustively all possible executions of a system, like, e.g., model checking, monitoring offers a number of advantages, including: (i) it can be applied directly on the implementation of the system, avoiding the risk of modeling errors; (ii) monitoring algorithms are usually very fast. However, an important limitation, which severely affects the effectiveness of monitoring, remains. Modern systems possess such a level of complexity that it is impossible for a system engineer to specify in advance all properties to be monitored. Moreover, even when this is possible, minor changes to the system may introduce unforeseen bugs.

In this paper, we investigate how to solve this problem by pairing monitoring with machine learning, which is used to learn in an iterative fashion new formulas to monitor by analysing trace prefixes that lead to failure. The method that we propose has three distinguishing features:

- *interpretability*: the machine learning methods that we use manipulate and produce only STL formulae, that can be easily inspected by a system engineer;
- formal guarantees on *monitorability*: every learned formula is guaranteed to be monitorable;
- *expressiveness*: the language for the specification of properties is proved to be able to express more properties than languages typically used in this context.

Our contributions are the following. First, we focus on the specification language to be used to specify properties. We introduce G(ppSTL) and F(ppSTL), the syntactic *safety* and *cosafety* fragments of STL, respectively. A formula of G(ppSTL) is of the form G(α) where G is the *globally* operator (which forces α to be always true) and α is a formula looking only to the past. Therefore, formulas of G(ppSTL) are used to express conditions that has to hold always, *i.e.*, invariants. Similarly, F(ppSTL) is used to express properties that should hold at least one time in the future, like, e.g., planning goals. A key feature of the two fragments is that they can look arbitrarily back into the past, a feature which is not offered by the fragments of STL that are commonly used in this context, in particular the *bounded future fragment of* STL (bfSTL). In the following, we formally prove that these fragments are *more expressive* than bfSTL.

We also give formal guarantees on the *monitorability* of the formulas of the fragments: we prove that each formula

Appendix and Supplementary materials are available at: https://github.com/dslab-uniud/ppSTL-IJCAI2024 in $\mathsf{G}(\mathsf{ppSTL})$ and $\mathsf{F}(\mathsf{ppSTL})$ is monitorable. On the one hand, this avoids the risk of generating nonmonitorable formulas in the learning phase that we will describe later; on the other hand, checking monitorability, which can be cumbersome [Havelund and Peled, 2023], is no more necessary. In addition, for the case of formulas interpreted over qualitative time, *i.e.*, system's executions with no timestamps attached to each time point, we prove that $\mathsf{G}(\mathsf{ppSTL})$ and $\mathsf{F}(\mathsf{ppSTL})$ capture the *entire* class of safety and cosafety properties, which form an important subclass of monitorable properies.

Third, we devise a framework for the automatic discovery of relevant properties, written in $\mathsf{G}(\mathsf{ppSTL})$ and $\mathsf{F}(\mathsf{ppSTL})$, based on traces that lead to failure. The objective here is to generate a pool of formulas that helps *anticipating* the identification of failures, a task carried out by a multi-objective genetic algorithm. The results of our experiments show that the pool of formulas obtained in this way can be effectively used for anticipating the detection of failures. Moreover, they can be easily checked by a system engineer, making the *interpretability* of results a distinguished feature of our method. In a dedicated section, we discuss the differences of our methodology with respect to existing solutions for the integration of monitoring and machine learning.

The paper is structured as follows. In Section 2, we provide some background knowledge. Section 3 is dedicated to the safety and cosafety fragments of STL and to their theoretical properties (monitorability and expressiveness). In Section 4, we describe our methodology to learn new formulas based on genetic programming. The outcomes of the experimental evaluation are reported in Section 5. In Section 6, we discuss the work done, highlight its strength and some limitations still present, and provide future research directions.

2 Background

2.1 Signal and Metric Temporal Logic

Let $\mathbb{T}$ be a set of *timestamps*, which are numbers attached to each time point that represent the (real) time at which an event has occurred. There are several possible choices for $\mathbb{T}$, depending on how time instants are modeled, mainly: (i) $\mathbb{T} \coloneqq \mathbb{N}$, for *qualitative-time*; (ii) $\mathbb{T} \coloneqq \mathbb{R}$, for *real-time*.

Syntax

From now on, given a set of variables $x_1, \dots, x_n$, we denote by $\mathbb{D}$ their domain. We define the syntax of *Signal Temporal Logic* [Maler and Nickovic, 2004] as follows.

Definition 1 (Signal Temporal Logic)**.** *Formulas ϕ of* Signal Temporal Logic *(*STL*, for short) are inductively defined as follows:*

$$\phi \coloneqq f_i(\overline{x}) \otimes c \mid \neg\phi \mid \phi \vee \phi \mid \mathsf{X}\phi \mid \phi\, \mathsf{U}_I\, \phi \mid \mathsf{Y}\phi \mid \phi\, \mathsf{S}_I\, \phi$$

where $\overline{x} \coloneqq \langle x_1, \dots, x_n\rangle$ for some $n \in \mathbb{N}$, each variable x_j, for $1 \leq j \leq n$, takes value in the set $\mathbb{D}$, $f_i : \mathbb{D}^n \to \mathbb{D}$ is a computable function, $c \in \mathbb{D}$, and $\otimes \subseteq \mathbb{D} \times \mathbb{D}$. In addition, we require I to be an interval of the form $[a, b]$, with $a \leq b$, or $[a, \infty)$, where $a, b \in \mathbb{T}$ are two timestamps represented in binary *notation.*

Formulas of type $f_i(\overline{x}) \otimes c$ are called *atomic formulas*. Modalities U_I and S_I are called *until* and *since*, respectively. We define the standard shortcut operators as follows: (i) *true*: $\top \coloneqq f_i(\overline{x}) \vee \neg f_i(\overline{x})$; (ii) *weak tomorrow*: $\widetilde{\mathsf{X}}\phi \coloneqq \neg\mathsf{X}\neg\phi$; (iii) *eventually*: $\mathsf{F}_I\phi \coloneqq \top\, \mathsf{U}_I\, \phi$; (iv) *globally*: $\mathsf{G}_I\phi \coloneqq \neg\mathsf{F}_I\neg\phi$; (v) *release*: $\phi_1\, \mathsf{R}_I\, \phi_2 \coloneqq \neg(\neg\phi_1\, \mathsf{U}_I\, \neg\phi_2)$; (vi) *weak yesterday*: $\widetilde{\mathsf{Y}}\phi \coloneqq \neg\mathsf{Y}\neg\phi$; (vii) *once*: $\mathsf{O}_I\phi \coloneqq \top\, \mathsf{S}_I\, \phi$; (viii) *historically*: $\mathsf{H}_I\phi \coloneqq \neg\mathsf{O}_I\neg\phi$; (ix) *triggers*: $\phi_1\, \mathsf{T}_I\, \phi_2 \coloneqq \neg(\neg\phi_1\, \mathsf{S}_I\, \neg\phi_2)$.

When $I = [0, \infty)$, we say that U_I (resp., S_I) is *unbounded* and we simply write U (resp., S); otherwise, it is *bounded*. The same holds for the other operators. Modalities X, U_I, and all shortcuts derived from them, are called *future modalities*, while Y, S_I, and all shortcuts derived from them, are called *past modalities*. We denote by STL the set of STL formulas.

The *bounded future fragment of* STL (denoted by bfSTL) is the set of STL formulas such that each of its temporal modalities is bounded by an interval of the form $[a, b]$, with $a, b \in \mathbb{T}$.

Metric Temporal Logic (denoted by MTL) is the set of formulas obtained from the same grammar as STL, but by replacing the base case, *i.e.*, $f_i(\overline{x}) \otimes c$, by an atomic proposition p taken from a set $\mathcal{AP}$.

Semantics

From now on, we fix a set of n variables $x_1, \dots, x_n$. A *real-time n-valued state* is a pair (t, v), where $t \in \mathbb{T}$ is a timestamp and $v \in \mathbb{D}^n$ represents the n values $d_1, \dots, d_n$ for the variables $x_1, \dots, x_n$, respectively. A *real-time n-valued trace σ* is a sequence of real-time n-valued states: σ is *infinite* when $\sigma \in (\mathbb{T} \times \mathbb{D}^n)^\omega$ and *finite* when $\sigma \in (\mathbb{T} \times \mathbb{D}^n)^*$. We often write σ as the sequence $\langle \sigma_0, \sigma_1, \dots\rangle$. We denote by $|\sigma|$ the *length of σ*, *i.e.*, the number of its states; if σ is infinite, then we set $|\sigma| = \omega$. For any $0 \leq i < |\sigma|$, we denote by $\sigma_{[0,i]}$ the *prefix* of σ up to position i. For all $\sigma \in (\mathbb{T} \times \mathbb{D}^n)^*$ and all $\sigma' \in (\mathbb{T} \times \mathbb{D}^n)^* \cup (\mathbb{T} \times \mathbb{D}^n)^\omega$, we denote by $\sigma \cdot \sigma'$ the trace obtained by concatenating σ' to σ.

We interpret STL formulas with variables $x_1, \dots, x_n$ over *infinite real-time n-valued* traces $\sigma = \langle \sigma_0, \sigma_1, \dots, \sigma_i, \dots\rangle$ that have to satisfy the following conditions:

- *(strict) monotonicity*: for all $i, j \in \mathbb{N}$, with $i < j$, if $\sigma_i = (t, v)$ and $\sigma_j = (t', v')$, then $t < t'$;
- *progress*: for all $n \in \mathbb{N}$, there exists $i \in \mathbb{N}$ such that $\sigma_i = (t, v)$ and $n < t$.

A real-time n-valued language $\mathcal{L}$ is a set of real-time n-valued traces. By $\overline{\mathcal{L}}$ we denote the *complement* of $\mathcal{L}$, that is, $\overline{\mathcal{L}} \coloneqq \{\sigma \in (\mathbb{T} \times \mathbb{D}^n)^\omega \mid \sigma \notin \mathcal{L}\}$. We say that $\mathcal{L}$ is a *real-time multi-valued language* if and only if it is a real-time n-valued language, for some $n \in \mathbb{N} \setminus \{0\}$.

Given an STL formula ϕ with variables $x_1, \dots, x_n$, an *infinite real-time n-valued* trace $\sigma \in (\mathbb{T} \times \mathbb{D}^n)^\omega$, and a position i, with $i \geq 0$, we define the *satisfaction of ϕ over σ at position* i, written $\sigma, i \models \phi$, inductively as follows:

1. $\sigma, i \models f_i(\overline{x}) \otimes c$ iff $\sigma_i = (t, (v_1, \dots, v_n))$ and $f_i(v_1, \dots, v_n) \otimes c$ is true;
2. $\sigma, i \models \neg\phi$ iff $\sigma, i \not\models \phi$;
3. $\sigma, i \models \phi_1 \vee \phi_2$ iff $\sigma, i \models \phi_1$ or $\sigma, i \models \phi_2$;
4. $\sigma, i \models \mathsf{X}\phi_1$ iff $\sigma, i + 1 \models \phi_1$;
5. $\sigma, i \models \phi_1\, \mathsf{U}_{[a,b]}\, \phi_2$ iff there exists a $j \geq i$ such that: (i) $\sigma, j \models \phi_2$ and $t + a \leq t' \leq t + b$, where $\sigma_i = (t, v)$

and $\sigma_j = (t', v')$, and (ii) for all $i \leq k < j$, it holds that $\sigma, k \models \phi_1$;

6. $\sigma, i \models \mathsf{Y}\phi_1$ iff $i > 0$ and $\sigma, i-1 \models \phi$;
7. $\sigma, i \models \phi_1 \mathsf{S}_{[a,b]} \phi_2$ iff there exists a $0 \leq j \leq i$ such that: (i) $\sigma, j \models \phi_2$ and $t - b \leq t' \leq t - a$, where $\sigma_i = (t, v)$ and $\sigma_j = (t', v')$, and (ii) for all $j < k \leq i$, it holds that $\sigma, k \models \phi_1$;

Formulas of STL are interpreted at the *first* position of the trace σ: we say that σ *is a model of the* STL *formula* ϕ (written $\sigma \models \phi$) iff $\sigma, 0 \models \phi$. We define the language of an STL formula as follows.

Definition 2 (Language of an STL formula)**.** *Given an* STL *formula* ϕ *with variables* $x_1, \ldots, x_n$, the language of ϕ*, denoted by* $\mathcal{L}(\phi)$*, is the real-time* n*-valued language* $\{\sigma \in (\mathbb{T} \times \mathbb{D}^n)^\omega \mid \sigma \models \phi\}$.

The semantics of MTL is defined by considering infinite real-time traces in $(\mathbb{T} \times 2^{AP})^\omega$, for a given set of atomic propositions $\mathcal{AP}$, instead of real-time multi-valued traces, like for STL. The base case of the semantics is defined as follows:

1. $\sigma, i \models p$ iff $\sigma_i = (t, v)$ and $p \in v$.

The rest of the semantic clauses remain the same as in the case of STL. The language of an MTL formula ϕ over the propositional atoms $\mathcal{AP}$, denoted by $\mathcal{L}(\phi)$, is defined as $\mathcal{L}(\phi) := \{\sigma \in (\mathbb{T} \times 2^{\mathcal{AP}})^\omega \mid \sigma, 0 \models \phi\}$.

2.2 Safety and Cosafety Fragments

Cosafety real-time multi-valued languages are defined as follows.

Definition 3 (Cosafety real-time multi-valued language)**.** *Let* $\mathcal{L} \subseteq (\mathbb{T} \times \mathbb{D}^n)^\omega$ *be a real-time multi-valued language. We say that* $\mathcal{L}$ *is a* cosafety language *iff, for all* $\sigma \in (\mathbb{T} \times \mathbb{D}^n)^\omega$*, if* $\sigma \in \mathcal{L}$*, then there exists* $i \geq 0$ *such that* $\sigma_{[0,i]} \cdot \sigma' \in \mathcal{L}$*, for all* $\sigma' \in (\mathbb{T} \times \mathbb{D}^n)^\omega$.

Safety real-time multi-valued languages are defined as the duals of cosafety ones.

Definition 4 (Safety real-time multi-valued language)**.** *Let* $\mathcal{L} \subseteq (\mathbb{T} \times \mathbb{D}^n)^\omega$ *be a real-time multi-valued language. We say that* $\mathcal{L}$ *is a* safety language *iff* $\overline{\mathcal{L}}$ *is a cosafety real-time multi-valued language.*

2.3 Monitoring

Monitoring is a lightweight runtime verification technique [Leucker and Schallhart, 2009] and it consists of the generation of a *monitor* that checks an execution of a system either in an online fashion (*i.e.*, at runtime) or in an offline fashion (*e.g.*, by analysing the log of the system). A monitor reports two types of results: an inconclusive output (denoted by $?$) or an irrevocable verdict, in particular a violation (resp., a satisfaction) in case all the continuations of that execution are bad (resp., good). We define a *monitor for a language* $\mathcal{L} \subseteq (\mathbb{T} \times \mathbb{D}^n)^\omega$ as a function $\mathrm{mon}_{\mathcal{L}} : (\mathbb{T} \times \mathbb{D}^n)^* \to \{\top, \bot, ?\}$ such that, for all $\sigma \in (\mathbb{T} \times \mathbb{D}^n)^*$,

$$\mathrm{mon}_{\mathcal{L}}(\sigma) := \begin{cases} \top \text{ iff } \forall \sigma' \in (\mathbb{T} \times \mathbb{D}^n)^\omega \,.\, \sigma \cdot \sigma' \in \mathcal{L} \\ \bot \text{ iff } \forall \sigma' \in (\mathbb{T} \times \mathbb{D}^n)^\omega \,.\, \sigma \cdot \sigma' \notin \mathcal{L} \\ ? \text{ otherwise} \end{cases}$$

Given an STL formula ϕ, we will denote with mon_ϕ the monitor $\mathrm{mon}_{\mathcal{L}(\phi)}$. When the monitor returns $\top$, we know that all continuations are good w.r.t. the property ϕ: this is the case, e.g., of planning problems [Ghallab *et al.*, 2004], where ϕ expresses the achievement of a goal. If the monitor returns $\bot$, we know that there has been an irremediable violation of ϕ: this is the case, for instance, of invariance properties, requiring that something bad never happens [Kupferman and Vardi, 2001].

We point out the similarity between the $\top$ (resp., $\bot$) result of monitors and cosafety (resp., safety) properties. In fact, all cosafety properties and all safety properties are *monitorable*, in the sense that, for every trace σ, there exists at least one continuation σ' such that $\mathrm{mon}_\phi(\sigma \cdot \sigma') \in \{\top, \bot\}$. We define *monitorability* as follows.

Definition 5 (Monitorability)**.** *For each* $\mathcal{L} \subseteq (\mathbb{T} \times \mathbb{D}^n)^\omega$*, we say that* $\mathcal{L}$ *is* monitorable *iff, for all* $\sigma \in (\mathbb{T} \times \mathbb{D}^n)^*$*, there exists a* $\sigma' \in (\mathbb{T} \times \mathbb{D}^n)^*$*, such that* $\mathrm{mon}_\phi(\sigma \cdot \sigma') \neq ?$.

We say that an STL formula ϕ is monitorable iff $\mathcal{L}(\phi)$ is monitorable. Not all STL formulas are monitorable. As an example, the formula $\mathsf{G}(x > 0 \to \mathsf{F}y < 0)$ stating that every time x is greater than 0 there exists a point in the future where y is negative, is *not* monitorable. However, every (co)safety property is monitorable.

Proposition 1 ([Bauer *et al.*, 2011])**.** *For every* $\mathcal{L} \subseteq (\mathbb{T} \times \mathbb{D}^n)^\omega$*, if* $\mathcal{L}$ *is safety or cosafety, then* $\mathcal{L}$ *is monitorable.*

We point out that the *vice versa* of Proposition 1 does not hold: there exist monitorable languages that are neither safety nor cosafety [Bauer *et al.*, 2011].

3 Monitorable Fragments of STL

In this section, we define the safety and the cosafety syntactic fragments of STL. We show that they express, respectively, only safety and cosafety languages, and thus only *monitorable* properties. In addition, in the case of *qualitative time*, we prove also the *vice versa*, *i.e.*, all safety and all cosafety languages definable in STL can be defined in (one of) the two fragments. The proof of all lemmas and theorems can be found in the Appendix A.

3.1 The $\mathsf{G}(\mathsf{ppSTL})$ and the $\mathsf{F}(\mathsf{ppSTL})$ Fragments

The safety and the cosafety syntactic fragments of STL are based on the *pure past fragment of* STL, which comprises formulas of STL that can look only into the past, and is defined as follows.

Definition 6 (The pure past fragment of STL)**.** *The* pure past fragment of STL*, denoted by* ppSTL*, is the set of* STL *formulas devoid of future operators.*

Unlike the case of STL formulas, in the case of ppSTL we consider only *finite, nonempty* real-time multi-valued traces. Formulas of ppSTL are interpreted at the *last* position of a trace $\sigma \in (\mathbb{T} \times \mathbb{D}^n)^+$, and we say that σ *is a model of the* ppSTL *formula* ϕ iff $\sigma, |\sigma| - 1 \models \phi$.

We define the syntactic safety and cosafety fragments of STL as follows.

Definition 7 (Safety and Cosafety syntactic fragments of STL)**.** *We define the safety (resp., cosafety) syntactic fragment of* STL*, denoted by* $\mathsf{G(ppSTL)}$ *(resp.,* $\mathsf{F(ppSTL)}$*), as the set of* STL *formulas of the form* $\mathsf{G}(\psi)$ *(resp.,* $\mathsf{F}(\psi)$*), where* ψ *is a formula of* ppSTL.

There are three distinguishing features of the syntax of $\mathsf{G(ppSTL)}$ and $\mathsf{F(ppSTL)}$ that are worth pointing out:

1. The use of *past modalities*, which allows one to avoid the risk of considering *non-monitorable* formulas.
2. The use of *unbounded intervals*, which allows formulas of these fragments to constrain arbitrarily long (yet finite) traces, in contrast to formulas of the bounded fragment of STL which are able to constrain only finite and *bounded* portions of a trace.
3. The use multi-variable functions. In [Brunello *et al.*, 2023], only functions with arity 1 were allowed (*e.g.*, $x + 1 \leq 3$). Here, we deal with multi-variable functions to allow, for instance, the specification of constraints of the form $|x_1 - x_2| > 0$. This is in line with [Maler and Nickovic, 2004; Donzé and Maler, 2010].

Examples

Here, we give some examples of $\mathsf{G(ppSTL)}$ and $\mathsf{F(ppSTL)}$ formulas. They concern an *arbiter* that has to give some resources (grants) to the processes that make a request. We suppose to have two variables x_g and x_r, with domain $\mathbb{D} := \mathbb{N}$, that are set to a value strictly greater than 0 iff the arbiter gives a grant and the processes perform a request, respectively.

The simple requirement that *"a grant is always preceded by a request in at least* 2.1 *and at most* 7.4 *time units"* is captured by the formula $\phi := \mathsf{G}(x_g > 0 \to \mathsf{O}_{[2.1,7.4]}(x_r > 0))$. Suppose we want to express the *unbounded* version of the previous requirement, that is, *"a grant is always preceeded by a request"*. The $\mathsf{G(ppSTL)}$ formula for this requirement is $\mathsf{G}(x_g > 0 \to \mathsf{YO}(x_r > 0))$.

The requirement *"there is at least one request followed by a grant "* is modelled by the $\mathsf{F(ppSTL)}$ formula $\mathsf{F}(x_g > 0 \wedge \mathsf{YO}x_r > 0)$. Other examples are reported in the Appendix B.

Comparison with bfSTL

In the following, we prove that $\mathsf{G(ppSTL)}$ and $\mathsf{F(ppSTL)}$ are more expressive than bfSTL (the bounded fragment). The rationale is based on the fact that, while bfSTL formulas can constrain only bounded intervals of a trace, $\mathsf{G(ppSTL)}$ and $\mathsf{F(ppSTL)}$ can do the same but also over intervals of unbounded length. To show it, we first prove that ppSTL is more expressive than bfSTL.

Proposition 2 (ppSTL is more expressive than bfSTL)**.** *There exists a language* $\mathcal{L} \subseteq (\mathbb{T} \times \mathbb{D}^n)^*$ *(for some* $n \in \mathbb{N}$*) such that:*

- *there exists a* ppSTL *formula* ϕ *such that* $\mathcal{L}(\phi) = \mathcal{L}$*;*
- *there exists no* bfSTL *formula* ψ *such that* $\mathcal{L}(\psi) = \mathcal{L}$.

We define $\mathsf{G(bfSTL)}$ as the set of formulas of the form $\mathsf{G}(\alpha)$, where $\alpha \in \mathsf{bfSTL}$, and we define $\mathsf{F(bfSTL)}$ analogously. From the previous proposition, and from the fact that $\mathsf{G(bfSTL)}$ and $\mathsf{F(bfSTL)}$ are syntactic fragments of $\mathsf{G(ppSTL)}$ and $\mathsf{F(ppSTL)}$, respectively, it follows that:

- $\mathsf{G(ppSTL)}$ is strictly more expressive than $\mathsf{G(bfSTL)}$;
- $\mathsf{F(ppSTL)}$ is strictly more expressive than $\mathsf{F(bfSTL)}$.

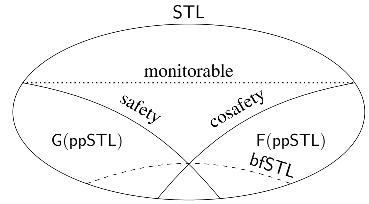


Figure 1: Summary of $\mathsf{G(ppSTL)}$ and $\mathsf{F(ppSTL)}$ expressive power for qualitative time.

Monitorability of $\mathsf{G(ppSTL)}$ **and** $\mathsf{F(ppSTL)}$

In the following, we show that $\mathsf{G(ppSTL)}$ (resp., $\mathsf{F(ppSTL)}$) expresses only safety (resp., cosafety) real-time multi-valued languages (*cf.* Definitions 3 and 4).

Lemma 1. *For all* $\phi \in \mathsf{G(ppSTL)}$ *(resp.,* $\phi \in \mathsf{F(ppSTL)}$*), it holds that* $\mathcal{L}(\phi)$ *is a safety (resp., cosafety) real-time multi-valued language.*

It follows from Proposition 1 that all properties definable in $\mathsf{G(ppSTL)}$ or in $\mathsf{F(ppSTL)}$ are *monitorable*.

Theorem 1. *For all* $\phi \in \mathsf{G(ppSTL)} \cup \mathsf{F(ppSTL)}$*, it holds that* $\mathcal{L}(\phi)$ *is monitorable.*

In addition, we prove that, when interpreted over *qualitative time*, the fragments $\mathsf{G(ppSTL)}$ and $\mathsf{F(ppSTL)}$ are expressively *complete*, that is, every safety (resp., cosafety) multi-valued language that can be defined by an STL formula is definable also in $\mathsf{G(ppSTL)}$ (resp., in $\mathsf{F(ppSTL)}$).

Theorem 2 (Expressive Completeness over qualitative-time)**.** *For all multi-valued languages* $\mathcal{L} \subseteq (\mathbb{N} \times \mathbb{D}^n)^*$ *definable in* STL*, it holds that:*

- $\mathcal{L}$ *is safety* iff *there exists a formula* ϕ *of* $\mathsf{G(ppSTL)}$ *such that* $\mathcal{L}(\phi) = \mathcal{L}$*;*
- $\mathcal{L}$ *is cosafety* iff *there exists a formula* ϕ *of* $\mathsf{F(ppSTL)}$ *such that* $\mathcal{L}(\phi) = \mathcal{L}$.

We point out that, to the best of our knowledge, it is unknown whether Theorem 2 holds for the case of real-time. A recap of the expressiveness results is given in Fig. 1.

4 Failure Detection Framework

In this section, we describe the proposed framework and the formula learning algorithm that it uses.

4.1 The Overall Framework

Algorithm 1 describes the framework training phase. It monitors, one after the other, all available failure system traces and, for each one, it simulates its point-by-point arrival (i.e., all the prefixes of the trace). At the end, it returns a pool of formulas $\mathcal{P}$ characterizing bad behaviours of the system.

The procedure gets, as its input, a pool $\mathcal{P}$ of formulas encoding bad behaviours. The pool may be empty, or it may already include some formulas, if they were previously defined by domain experts. In addition, a training set $\mathcal{X}$ is provided, consisting of pairs $(\sigma, is\_failure)$, where σ represents a system execution trace and $is\_failure$ its corresponding label

Algorithm 1: Framework training phase

input: $\mathcal{P}$ initial (possibly empty) pool of formulas,
$\mathcal{X}$ dataset of labelled traces $(\sigma, is\_failure)$,
q quality requirements

1: $\Sigma_\top \leftarrow \{\sigma \mid (\sigma, is\_failure) \in \mathcal{X} \wedge is\_failure = \top\}$
2: $\Sigma_\bot \leftarrow \{\sigma \mid (\sigma, is\_failure) \in \mathcal{X} \wedge is\_failure = \bot\}$
3: $\Sigma_\bot \leftarrow$ GENAUGMENTEDTRACES$(\Sigma_\bot)$
4: **for** $\sigma \in \Sigma_\top$ **do**
5:     $failure\_detected \leftarrow \bot$
6:     **for** $i \leftarrow 0$ **to** $|\sigma| - 1$ **do**
7:         $\mathcal{F} \leftarrow \{\psi \in \mathcal{P} \mid \mathrm{mon}_\psi(\sigma_{[0,i]}) = \bot\}$
8:         **if** $\mathcal{F} \neq \varnothing$ **then**
9:             $failure\_detected \leftarrow \top$
10:             $\mathcal{T} \leftarrow$ GENAUGMENTEDTRACES$(\sigma_{[0,i]})$
11:             $\Phi \leftarrow$ LEARNFORMULA$(\mathcal{T}, \Sigma_\bot, q)$
12:             $\mathcal{P} \leftarrow \mathcal{P} \cup \Phi$
13:             **break**
14:         **end if**
15:     **end for**
16:     **if** $failure\_detected = \bot$ **then**
17:         $\mathcal{T} \leftarrow$ GENAUGMENTEDTRACES(σ)
18:         $\Phi \leftarrow$ LEARNFORMULA$(\mathcal{T}, \Sigma_\bot, q)$
19:         $\mathcal{P} \leftarrow \mathcal{P} \cup \Phi$
20:     **end if**
21: **end for**
22: **return** $\mathcal{P}$

($\top$, if σ is a trace ending with a failure; $\bot$ otherwise).[1] Finally, quality requirements q are considered that should be satisfied by any new formula extracted during the training process. They will be described more in detail later.

On Line 1 (resp., Line 2) the subset of failure (resp., good) traces is extracted from $\mathcal{X}$. For each good trace, n_{aug} variants (a framework global parameter) are generated by adding random Gaussian noise as a counter-overfitting measure. As the result, a set of $|\Sigma_\bot| + n_{aug} * |\Sigma_\bot|$ traces is obtained.

At this point, the framework starts its iterative part, during which a failure trace σ is monitored sequentially, point-by-point (Lines 6–15). At each iteration, the framework restricts its attention to the prefix $\sigma_{[0,i]}$ of trace σ, and it computes the set $\mathcal{F}$ of formulas leading to a *violation* (Line 7). To such an extent, it executes the monitoring tool `rtamt` [Ničković and Yamaguchi, 2020]. Since all formulas $\psi \in \mathcal{P}$ are meant to encode bad behaviors, we say that a formula ψ leads to a violation if $\mathrm{mon}_\psi(\sigma_{[0,i]}) = \bot$ (mon is defined as in Section 2.3).

If at least one violation is detected, data to be used for the extraction of a new formula are generated by the function GENAUGMENTEDTRACES (Line 10). Again, the execution trace $\sigma_{[0,i]}$ is perturbed by adding random Gaussian noise as a counter-overfitting measure, thus producing a set of augmented traces $\mathcal{T}$ of cardinality $n_{aug} + 1$.

Next, function LEARNFORMULA (Line 11) tries to extract a set of formulae Φ able to identify a bad behaviour of the system that anticipates, over the traces in $\mathcal{T}$, the violation detected by the monitor at time instant i. In our case, the extraction is carried out by the genetic algorithm described in Section 4.2, limiting to maximum one formula ϕ, i.e., $|\Phi| \leq 1$. Notice that Φ may be *empty*, an event that occurs if no formula satisfying the quality criteria q can be extracted. The quality criteria relate to the actual ability of a given formula $\phi \in \Phi$ to elicit an anticipatory bad behavior from the traces $\mathcal{T}$, while maintaining a False Alarm Rate (the fraction of false failure detections, FAR) that does not exceed a specified threshold for the traces in $\Sigma_\bot$. In setting such a threshold, it should be considered that formulas with a high FAR may cause a degradation of the monitoring pool, where other ill-founded formulas are added as a result of their triggering.

The monitoring executed over all prefixes of σ ends when either σ is correctly recognized as a failure trace by a formula in $\mathcal{P}$ (`break` instruction on Line 13), or σ has run out of points without any failure detection. In the latter case, since trace σ was a failure one, we force the formula extraction process (Lines 15–19). This approach is inspired by the *teacher forcing* technique in deep learning [Williams and Zipser, 1989]. Initially, the framework starts with a possibly empty pool $\mathcal{P}$ of properties. In the extreme case $\mathcal{P} = \varnothing$, it cannot identify any bad behavior of the system and the pool update process is forcibly triggered by the code in Lines 15–19. This is akin to having an oracle guiding the framework. Over time, as $\mathcal{P}$ expands, it is expected to gradually replace the oracle's function in identifying failures.

During Algorithm 1 operation, the pool of formulas $\mathcal{P}$ is iteratively refined by adding new formulas which ought to predict bad behaviors earlier and with increased reliability and coverage. 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.

4.2 Genetic Programming Algorithm

The function LEARNFORMULA is realized by means of a genetic algorithm implementing a genetic programming task, the idea being to evolve formulas starting from an initial population of random solutions [Poli *et al.*, 2008].

In practice, we utilize a multi-objective evolutionary algorithm due to its inherent flexibility. This approach enables us to define a specific grammar that the formulas must adhere to, and it also facilitates the straightforward setup and adjustment of optimization objectives for their extraction. The algorithm is implemented through the library DEAP (Distributed Evolutionary Algorithms in Python) [Fortin *et al.*, 2012].

Figure 2 reports its intuitive operation. The algorithm gets in input the set $\mathcal{T}$ of augmented traces generated by `genAugmentedTraces`, each of length t_l, the set of augmented good training traces $\Sigma_\bot$, and the set of quality requirements q that the output formula has to satisfy. These include, as we will see, min_{acc} and max_{far}; they are related to respectively $\mathcal{T}$ and $\Sigma_\bot$. As for the result, as already mentioned, in our case the algorithm returns a set Φ composed of at most one logic formula such that it satisfies the quality requirements expressed in q and it captures an anticipatory bad behaviour exhibited by traces in $\mathcal{T}$.

[1] Since failures are terminating events, it is reasonable to assume that they can be detected as they occur. This allows to appropriately label the time series and to generate such a training set $\mathcal{X}$. The same assumption does not hold, e.g., for anomaly detection, a task which is not covered in this paper.

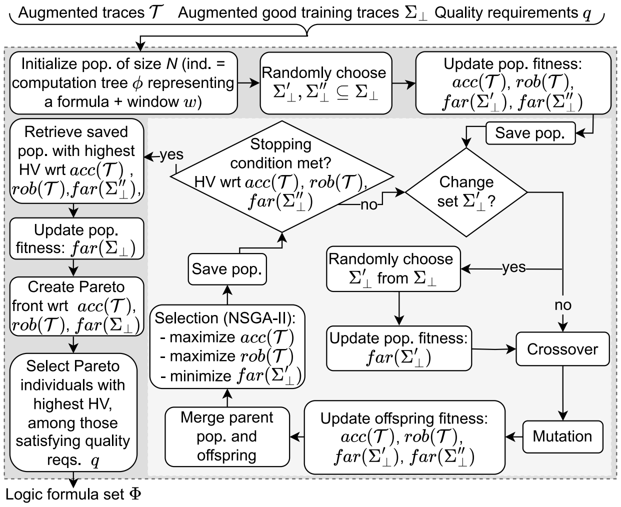


Figure 2: Overview of the genetic algorithm for formula extraction.

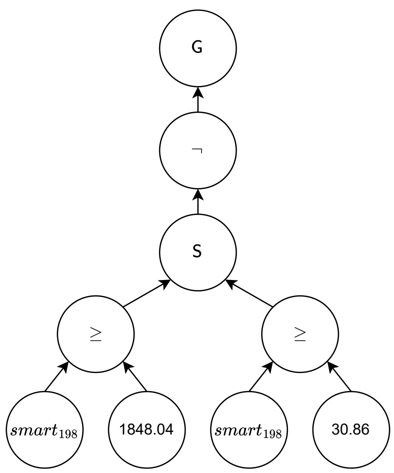


Figure 3: Computation tree of the formula $\mathsf{G}(\neg((smart_{198} \geq 1848.04)\,\mathsf{S}\,(smart_{198} \geq 30.86)))$.

Population and Its Initialization

As the first step, the population is initialized. Each individual of the population encodes a pair (ϕ, w), where ϕ is a computation tree representing either a bfSTL or a ppSTL formula (refer to Fig. 3 for an example) and w is an integer in the interval $[1, t_l - 1]$.

The computation tree ϕ is generated using DEAP's *genHalfAndHalf* method, configured to output a tree with a maximum height of 6, as recommended by Koza in his seminal work [Koza, 1994]. Specifically, half the time a tree whose leaves are all at the same depth is returned; in the remaining cases, different leaves may lay at different depths. In addition, care is taken to generate a population of individuals stratified with respect to the height of the trees.

The leaves of a computation tree may represent a variable v that refers to one of the signals that compose each multivariate trace $t \in \mathcal{T}$, or a constant $c \in [0, 1] \subseteq \mathbb{R}$. Note that, although the choice of the domain for the constants may seem restrictive, in fact, as its first operation, the genetic algorithm normalizes the input traces into the interval $[0, 1]$. Constants c and variables v are compared by means of the $\geq$ operator, in two manners: either a variable is compared with a constant $v \geq c$, or a variable is compared to another variable $v_1 \geq v_2$, where $v_1 \neq v_2$, i.e., they refer to two different signals.

Concerning the internal nodes of the tree, they encode logic operators following the syntax of either bfSTL or ppSTL. In addition, each temporal operator may be (always be, in the case of bfSTL formulas) paired with two interval bounds, i.e., $[a, b]$, with the constants $a, b \in \mathbb{N}$ and $a \leq b$. In generating a computation tree, idempotence is exploited to streamline the formula representation. This is achieved by the definition of a suitable grammar that avoids the combination of redundant operations, such as $\neg\neg\phi$, where ϕ is a bfSTL or a ppSTL formula.

All constants (including the individual's window w) are implemented through DEAP's *EphemeralConstant*.

As for w, it is used during the fitness evaluation step to partition each trace $t \in \mathcal{T}$ into a "good behaviour" and a "bad behaviour" sub-trace, as we will now discuss.

Fitness Function

Next, the fitness of each individual (ϕ, w) is updated along 4 dimensions. The first 2 pertain to the set of traces $\mathcal{T}$, while the latter 2 refer, respectively, to the sets $\Sigma'_\perp$ and $\Sigma''_\perp$. Both are randomly sampled from $\Sigma_\perp$, in a way such that $|\Sigma'_\perp| = |\Sigma''_\perp| = fract_{good} * |\Sigma_\perp|$ (where $fract_{good} \in [0, 1]$ is a global parameter of the framework).

As for the traces in $\mathcal{T}$, recall that they are all of the same length t_l and their last time instant corresponds to the one immediately preceding the failure detected within the framework training loop, either by means of a formula in the pool $\mathcal{P}$ or through teacher forcing. Thus, we would like the formula ϕ encoded by an individual to be able to elicit an anticipatory bad behaviour of that failure. To such an extent, each trace $t \in \mathcal{T}$ is divided into a good behaviour sub-trace $good(t) = t_{[0,w-1]}$ and a bad behaviour sub-trace $bad(t)$ as follows:

$$bad(t) = \begin{cases} t_{[w,t_l-1]} & \text{if formula } \phi \text{ is bfSTL} \\ t_{[0,w]} & \text{if formula } \phi \text{ is ppSTL} \end{cases} .$$

The idea is that the last time points of t are those closer to the detected failure, and thus should contain some prelude of it.

By means of `rtamt`, the monitor mon_ϕ is applied over all good and bad sub-traces, and the first fitness element, to be maximized, is obtained as follows:

$$acc(\mathcal{T}) = \frac{|\{\, t \mid t \in \mathcal{T} \wedge \mathrm{mon}_{\mathsf{G}\phi}(good(t)) \in \{\top, ?\}\,\}|}{|\mathcal{T}|} + \frac{|\{\, t \mid t \in \mathcal{T} \wedge \mathrm{mon}_{\phi}(bad(t)) = \perp\,\}|}{|\mathcal{T}|} .$$

Intuitively, $acc(\mathcal{T})$ represents the fraction of sub-traces correctly identified as good and failure ones; note how formula $\mathsf{G}\phi$ is considered on $good(t)$, as ϕ must be checked with respect to every time point in such traces.

The second fitness element, to be maximized, refers to the quantitative semantics of STL [Donzé and Maler, 2010]:

$$rob(\mathcal{T}) = min(rob_{good}, -rob_{bad}) ,$$

where rob_{good} (resp., rob_{bad}) is the average *robustness* of mon_ϕ calculated over all good (resp., bad) sub-traces. Due to the traces normalization step performed at the beginning of the genetic algorithm, in our case, both are real numbers in the interval $[-1, 1]$. Intuitively, the robustness of a formula ϕ with respect to a trace t tells us how far (or close) the formula is to be satisfied on a trace. For example, given $t_1 = [1, 0.9, 0.7]$, $t_2 = [0, 0.3, 0.4]$, and $\phi = \mathsf{G}(x \geq 0.5)$, the former trace has a robustness of 0.2, while the latter of -0.1. Note how the sign is switched on rob_{bad} to make it positive. Maximizing such a formula is akin to the concept of maximum margin separating hyperplane in the context of support vector machines [Hastie *et al.*, 2009].

As for set $\Sigma'_\perp$, again, `rtamt` is applied over all its traces, and the related third fitness element, to be minimized, intuitively represents the fraction of good traces of $\Sigma'_\perp$ in which a failure is incorrectly signalled. It is obtained as:

$$far(\Sigma'_\perp) = \frac{|\{\, \sigma \mid \sigma \in \Sigma'_\perp \wedge \mathrm{mon}_{\mathsf{G}\phi}(\sigma) = \perp \,\}|}{|\Sigma'_\perp|} \; .$$

Similarly, for the fitness element over $\Sigma''_\perp$, we calculate the fraction of good traces of $\Sigma''_\perp$ in which a failure is incorrectly signalled:

$$far(\Sigma''_\perp) = \frac{|\{\, \sigma \mid \sigma \in \Sigma''_\perp \wedge \mathrm{mon}_{\mathsf{G}\phi}(\sigma) = \perp \,\}|}{|\Sigma''_\perp|} \; .$$

Objectives $acc(\mathcal{T})$, $rob(\mathcal{T})$ and $far(\Sigma'_\perp)$ guide the evolutionary process, while $far(\Sigma''_\perp)$ is used to implement an early stopping strategy (which requires a fixed, i.e., not changing during the generations, reference set).

The initialization step of the algorithm finishes by saving the population along with its four fitness elements and the hypervolume calculated with respect to $acc(\mathcal{T})$, $rob(\mathcal{T})$ and $far(\Sigma''_\perp)$.

Let us now focus on the evolutionary part of the algorithm. Every $r_{interval}$ generations, the set $\Sigma'_\perp$ is resampled from $\Sigma_\perp$ as a counter-overfitting measure (see, for instance, [Gonçalves and Silva, 2013]). If a resample is performed, the fitness element $far(\Sigma'_\perp)$ is updated for each individual. Then, crossover and mutation operators are applied over the population, generating an offspring. Such operators are defined as follows.

Crossover

Let $i_1 = (\phi_1, w_1)$, $i_2 = (\phi_2, w_2)$ be two individuals from the population. From them, two other individuals can be created according to several crossover operations, randomly selected with uniform probability. The simplest one involves exchanging their windows, thus obtaining two new individuals $i'_1 = (\phi_1, w_2)$, $i'_2 = (\phi_2, w_1)$.

Also, the two computation trees corresponding to ϕ_1 and ϕ_2 can be hybridized by means of DEAP's *cxOnePointLeafBiased* operator. In short, it randomly selects a crossover point in each tree and exchanges the subtrees rooted in them. Again following suggestions by Koza [Koza, 1994], our operator is biased to choose the crossover point on internal nodes 90% of the times, while a leaf is chosen 10% of the times. Finally, we set a limit of 17 [Koza, 1994] on the generated individuals' height (DEAP's *staticLimit*). When an over-the-height-limit individual is generated, it is simply replaced by one of its parents, randomly selected.

Mutation

Given an individual $i = (\phi, w)$, also the mutation can be performed according to several strategies, chosen according to a uniform random probability. As for the window w, it can be re-generated by randomly choosing a number in the interval $[1, t_l - 1]$; or, it can be adjusted by a small random amount of points to the left or right, to allow for a finer tuning.

Concerning the computation tree corresponding to ϕ, three operations can be performed, taken from the DEAP primitives: *mutNodeReplacement*, which replaces a randomly chosen node by a randomly chosen operation with the same number of arguments; *mutShrink*, which shrinks the tree by choosing randomly a branch and replacing it with one of the branch's arguments (also randomly chosen); and, *mutEphemeral*, which randomly changes the values of all of the constants in the tree.

Once the offspring has been produced, their four previously defined fitness elements are determined. Next, the parent population and the offpring are merged, and a selection is performed relying on the classic strategy implemented in NSGA-II [Deb *et al.*, 2002], based on the concepts of *ranking* and *crowding distance*. Specifically, for each individual, the fitness elements $acc(\mathcal{T})$ (maximized), $rob(\mathcal{T})$ (maximized) and $far(\Sigma'_\perp)$ (minimized) are considered. The newly obtained population is then saved, along with its fitness elements and the hypervolume calculated with respect to $acc(\mathcal{T})$, $rob(\mathcal{T})$, and $far(\Sigma''_\perp)$.

The evolutionary part may end according to two conditions: either the maximum number of generations max_{gen} (global parameter of the framework) has been reached, or the early stopping condition is triggered. The latter is defined as follows: the population hypervolume, calculated with respect to $acc(\mathcal{T})$, $rob(\mathcal{T})$ and $far(\Sigma''_\perp)$ (i.e., the fixed set of good traces), is tracked along the generations. Then, if no improvement is observed for $patience$ generations, the execution is halted.

Recall that, during the execution of the algorithm, each generation's population has been saved. After the end of the iterative part of the evolutionary process, the population with the highest hypervolume with respect to $acc(\mathcal{T})$, $rob(\mathcal{T})$ and $far(\Sigma''_\perp)$ is recovered, and its individuals' false alarm rate with respect to all $\Sigma_\perp$ traces, $far(\Sigma_\perp)$, is established.[2] Then, the Pareto front of optimal solutions with respect to $acc(\mathcal{T})$, $rob(\mathcal{T})$ and $far(\Sigma_\perp)$ is extracted and filtered to keep only individuals satisfying the quality criteria q: $acc(\mathcal{T}) > min_{acc}$ and $far(\Sigma_\perp) \leq max_{far}$, where $min_{acc}, max_{far} \in [0, 1]$. Also, they are required to correctly classify the good and bad sub-traces originated from the original, unperturbed trace, always included in $\mathcal{T}$. Finally, among the remaining individuals, the formula of the best performing one is returned. This latter selection is based on choosing the individual with the highest hypervolume, calculated in relation to $acc(\mathcal{T})$, $rob(\mathcal{T})$, and $far(\Sigma_\perp)$.

[2] We use $\Sigma''_\perp$ instead of $\Sigma_\perp$ for computational efficiency.

4.3 Comparison To Related Work

Let us briefly contrast our framework with existing approaches to the integration of monitoring and learning.

Compared to contributions in the literature that extract temporal relations in time series data [Aggarwal *et al.*, 2018; Lu *et al.*, 2020; Petmezas *et al.*, 2021; Gao *et al.*, 2022], which are based on black-box techniques such as deep learning, our framework is highly interpretable (we refer the reader to Appendix E for some examples of extracted formulas). Interpretability is a paramount requirement in critical scenarios, such as healthcare and avionics. This is relevant especially in the light of recent results that underscore the issues behind feature attribution methods [Bilodeau *et al.*, 2024].

The closest proposal is the one described in [Brunello *et al.*, 2023], that uses genetic programming to learn new formulas of bfSTL. Here, we improved it under several respects.

From the theoretical point of view, we introduced the unbounded past variant of STL and defined the formalisms G(ppSTL) and F(ppSTL) which are strictly more expressive than bfSTL, that is, they can express more properties while still remaining in the realm of monitorability, removing the need of checking whether a newly learned formula is monitorable. In such respect, recall that, while bfSTL can only constrain bounded intervals of a trace, G(ppSTL) and F(ppSTL) do not inherit such a limitation, being able to constrain arbitrarily long intervals. Extending the learning algorithm of [Brunello *et al.*, 2023] to the new formalism was not trivial, and it advantageously led to the dismissal of the concept of *horizon*. The generation of monitorable-only formulas in the genetic algorithm is ensured by the design of a type-based grammar that allows only well-formed formulas adhering to the chosen formalism to be generated, while removing redundant operators (exploiting idempotence). In addition to these changes necessary to deal with the past, we introduced several new features:

- quality requirements for the new formulas are now clearly stated by means of q. Above all, it can be ensured that a formula's FAR (*i.e.*, the False Alarm Rate) does not exceed a specified threshold. This was not possible previously, resulting, over time, in a large number of formulas being removed from the pool after they had been learned;
- the imposed FAR threshold allowed us to redesign the offline learning phase of the framework increasing its efficiency. Now, it is sufficient to iterate only over failures, rather than the entire dataset;
- the refactored genetic algorithm makes use of a rotation-based strategy to counter overfitting on the good traces, includes an enriched set of crossover and mutation operations, and exploits an enhanced early stopping strategy, that returns the best population among the observed ones.

5 Experimental Evaluation

Here, we present the results of the application of the framework on three well-known datasets from the literature, comparing its performance with other recent contributions.

The *Backblaze Hard Drive* dataset contains information on the "health" status of hard drives, tracked by means of Self Monitoring Analysis and Reporting Technology (SMART). Each trace is described by: date of the report, serial number of the drive, a label indicating a drive failure, and 21 numerical SMART parameters. For comparison with the literature, we focus on [Brunello *et al.*, 2023] Split $S1$. The *Tennessee Eastman Process* (TEP) dataset is composed of simulated data from a fictitious chemical plant. Each trace has the features: trace ID, normal/faulty label, 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, we focus on the dataset FD001 and on the detection of the *Unhealthy state* class [Kim and Sohn, 2020]. Each simulation is represented by a multivariate time series, sampled at one value per second, obtained from 21 engine sensors. Details about the three datasets are included in Appendix C.

We instantiate two versions of the framework, using two different fragments of STL. The first, GP-bfSTL, relies on bfSTL formulas, as specified in the work by [Brunello *et al.*, 2023]. The second, GP-G(ppSTL), uses the more expressive G(ppSTL) fragment. Thereafter, we proceeded by first confirming the literature results by means of GP-bfSTL. Then, given such a baseline, we assess the performance of GP-G(ppSTL) to determine the contribution brought by the newly isolated logical fragment.

For each dataset, the GP-bfSTL framework is tuned considering a separate validation set obtained from training data (details are provided in Appendix D). Then, the best hyper parameters are used to run both the GP-bfSTL and GP-G(ppSTL) framework over all the training sets. Finally, performance is established by applying the monitor to classify each test set trace independently by means of the pool of properties obtained during training. To account for the inherent stochasticity of our approach, each experiment is repeated various times (of course, the same seeds are used with GP-bfSTL and GP-G(ppSTL)). Appendix F reports all the details about the code and how to use it to reproduce and use our framework for new research.

5.1 Results

Table 1 reports the main outcomes of the experiments; a discussion about the metrics employed as well as additional findings and analyses are in Appendix E. Note that the competitors' results are based on a single execution (still reported under the *Avg* columns), except for [Brunello *et al.*, 2023]. Thus, to allow for a fair comparison, for the F1 and MCC[3] metrics, which summarize the overall performance of the framework, we report not only the average but also the minimum and maximum values observed across the repetitions.

Overall, based on the F1 metric, which is provided by all the competitors, we can observe that GP-bfSTL achieved results on par with or superior to the considered baselines, confirming the soundness of our implementation. The GP-

[3]Matthews Correlation Coefficient, which is more informative than F1 in several scenarios [Chicco and Jurman, 2020].

Dataset	Approach	Prec.	Rec.	FAR	F1 Min	F1 Max	F1 Avg	MCC Min	MCC Max	MCC Avg
Backblaze $S1$	[Huang, 2017]	.51	.54	.00	-	-	.52	-	-	-
	[Lu *et al.*, 2020]	.87	.41	.00	-	-	.55	-	-	-
	[Brunello *et al.*, 2023]	.54	.60	.00	-	-	.56	-	-	-
	GP-bfSTL (our)	.76	.49	.00	.58	.61	.59	.58	.63	.61
	GP-G(ppSTL) (our)	.85	.47	.00	.55	.62	.60	.59	.64	.62
TEP	[Hajihosseini *et al.*, 2018]	1.0	1.0	-	-	-	1.0	-	-	-
	[Onel *et al.*, 2019]	1.0	1.0	.00	-	-	1.0	-	-	-
	[Brunello *et al.*, 2023]	1.0	1.0	.00	-	-	1.0	-	-	-
	GP-bfSTL (our)	1.0	1.0	.00	1.0	1.0	1.0	1.0	1.0	1.0
	GP-G(ppSTL) (our)	1.0	1.0	.00	1.0	1.0	1.0	.99	1.0	1.0
C-MAPSS	[Kim and Sohn, 2020]	.71	1.0	-	-	-	.83	-	-	-
	[Brunello *et al.*, 2023]	.96*	.77*	.01*	-	-	.86*	-	-	-
	GP-bfSTL (our)	.95	.65	.02	.70	.85	.77	.61	.80	.69
	GP-G(ppSTL) (our)	.98	.67	.01	.71	.87	.79	.63	.82	.73

Note: Results of [Lu *et al.*, 2020] listed as in [Brunello *et al.*, 2023]; the others are reported as in the original references. *: the original results in the referenced paper were based on an incorrectly generated test set of 143 traces, rather than the official set of 100 traces. The latter has been correctly considered in the present work.

Table 1: Experimental results

G(ppSTL) version, when run with the same hyper parameters as the GP-bfSTL one, scored in par or even better (cf. MCC), confirming the theoretical claims of Section 3. It is plausible that a dedicated tuning phase, which could not be performed due to time/resource limits, could enhance results further.

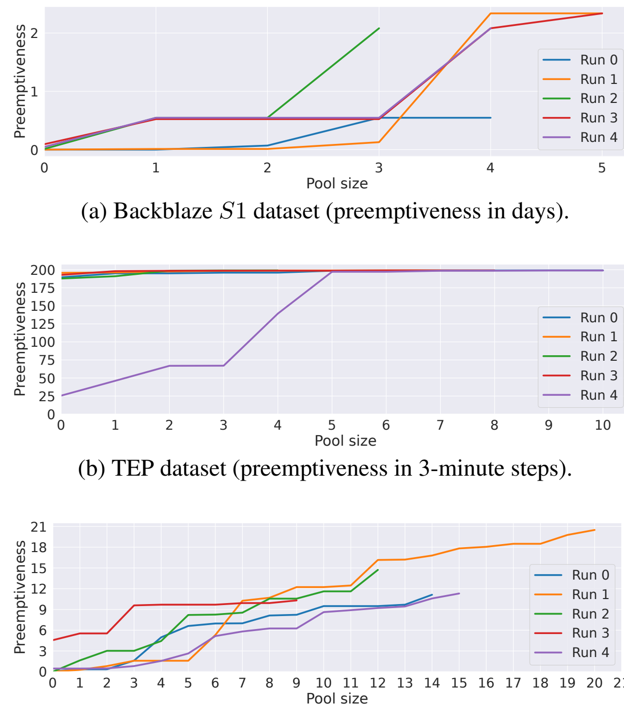
Note that a significant gap can sometimes be observed between the maximum/minimum performances of our framework. This stems from our chosen experimental workflow: for statistical soundness, also the framework's iterative learning process relies on a random shuffle of traces (Algorithm 1, Line 4), which in turn affects the formula extraction process. We speculate that learning on the training set through multiple passes until the pool converges (i.e., when no more properties are added) would yield more stable results.

Focusing on the GP-G(ppSTL) version of the framework, being based on the newly isolated fragment and the best performing one, Figure 4 illustrates how the average preemptiveness in failure detection on test set instances evolves as more and more formulas are learned and incorporated into the monitoring pool. In the dataset TEP, a plateau is reached around a value of 200, meaning that the formulas in the pool are capable of (perfectly, given Table 1) identifying a failure roughly 10 hours before it happens (recall the 3-minute sampling time used for the time series in this dataset). An increase in the anticipatory behaviour of the framework is observed also in the datasets C-MAPSS and Backblaze $S1$.

Finally, we report an example of formulas extracted during the framework training phase for the Backblaze $S1$ dataset.

Formula $f_1 = \mathsf{G}(smart_{187} < 24.40 \vee smart_1 \geq 169483444)$, included in $\mathcal{P}$, evaluates to false, thus issuing a failure detection. It means that the number of *reported uncorrectable errors* (sensor $smart_{187}$) exceeded the 24.40 threshold and, at the same time, the *read error rate* (sensor $smart_1$) was lower than 169483444. As a result, another formula, intended to anticipate the situation modeled by f_1, is extracted by the genetic algorithm and added to $\mathcal{P}$. It is $f_2 = \mathsf{G}(smart_{198} < 16.56)$, which is violated when the *uncorrectable sector count* exceeds the 16.56 threshold.

Thus, when viewed from the perspective of the domain, an event marked by the violation of f_2 can be regarded as the direct cause of the scenario projected by the violation of f_1. In other words, hardware errors (indicated by f_2's violation) are the root cause of subsequent read/write errors (as modeled by the violation of f_1).



(a) Backblaze $S1$ dataset (preemptiveness in days).

(b) TEP dataset (preemptiveness in 3-minute steps).

(c) C-MAPSS dataset (preemptiveness in seconds).

Figure 4: Average preemptiveness evolution on the test set instances, as more properties are learned and added to the pool, for 5 different runs of the framework, GP-G(ppSTL) case.

6 Conclusions and Discussion

In this paper, we propose to exploit machine learning to enhance monitoring of STL formulas. Our methodology is based on the identification of two fragments of STL, namely G(ppSTL) and F(ppSTL), for which we give formal guarantees on monitorability and expressiveness, and on the use of genetic programming to automatically learn new relevant formulas of these fragments. The experiments reveal that formulas generated in this way are effective in anticipating the discovery of failures.

We conclude by discussing future research directions, in particular about formula extraction and management. The use of a genetic algorithm facilitated rapid prototyping and offered flexibility. Still, more efficient methodologies should be explored. For example, the use of generative and/or reinforcement learning techniques [Holt *et al.*, 2023], or a hybrid approach combining deep learning with evolutionary computation, could offer substantial advancements [Chen *et al.*, 2018; Chen *et al.*, 2020; Mundhenk *et al.*, 2021; Qian *et al.*, 2021]. Graph neural networks are promising as well, as they are proven effective for performing symbolic regression tasks [Cranmer *et al.*, 2020], and may more effectively exploit formulas, e.g., considering their automaton or directed acyclic graph encodings. Finally, the framework itself should be expanded to support online continual learning and update of formulas, rather than relying solely on a fixed training dataset. This is crucial for real-world applications where monitored systems may change their behaviour over time.

Acknowledgments

All the authors acknowledge the support from the 2024 Italian INdAM-GNCS project "Certificazione, monitoraggio, ed interpretabilità in sistemi di intelligenza artificiale", ref. no. CUP E53C23001670001. Luca Geatti, Angelo Montanari, and Nicola Saccomanno also acknowledge the support from the Interconnected Nord-Est Innovation Ecosystem (iNEST), which received funding from the European Union Next-GenerationEU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) – MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.5 – D.D. 1058 23/06/2022, ECS00000043). In addition, Angelo Montanari acknowledges the support from the MUR PNRR project FAIR - Future AI Research (PE00000013) also funded by the European Union Next-GenerationEU. This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

References

[Aggarwal *et al.*, 2018] Karan Aggarwal, Onur Atan, Ahmed K Farahat, Chi Zhang, Kosta Ristovski, and Chetan Gupta. Two birds with one network: Unifying failure event prediction and time-to-failure modeling. In *Proceedings of IEEE International Conference on Big Data (Big Data)*, pages 1308–1317. IEEE, 2018.

[Bauer *et al.*, 2011] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):1–64, 2011.

[Bilodeau *et al.*, 2024] Blair Bilodeau, Natasha Jaques, Pang Wei Koh, and Been Kim. Impossibility theorems for feature attribution. *Proceedings of the National Academy of Sciences*, 121(2):e2304406120, 2024.

[Brunello *et al.*, 2023] Andrea Brunello, Dario Della Monica, Angelo Montanari, Nicola Saccomanno, and Andrea Urgolo. Monitors that learn from failures: Pairing STL and genetic programming. *IEEE Access*, 11:57349–57364, 2023.

[Chen *et al.*, 2018] Yukang Chen, Gaofeng Meng, Qian Zhang, Shiming Xiang, Chang Huang, Lisen Mu, and Xinggang Wang. Reinforced evolutionary neural architecture search. *arXiv preprint arXiv:1808.00193*, 2018.

[Chen *et al.*, 2020] Diqi Chen, Yizhou Wang, and Wen Gao. Combining a gradient-based method and an evolution strategy for multi-objective reinforcement learning. *Applied Intelligence*, 50:3301–3317, 2020.

[Chicco and Jurman, 2020] Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genomics*, 21(1):1–13, 2020.

[Cranmer *et al.*, 2020] M. Cranmer, Alvaro Sanchez-Gonzalez, P. Battaglia, Rui Xu, K. Cranmer, D. Spergel, and S. Ho. Discovering symbolic models from deep learning with inductive biases. *ArXiv*, abs/2006.11287, 2020.

[Deb *et al.*, 2002] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[Donzé and Maler, 2010] Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In Krishnendu Chatterjee and Thomas A. Henzinger, editors, *Proceedings of the 8th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2010)*, pages 92–106. Springer, 2010.

[Fortin *et al.*, 2012] 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*, 13:2171–2175, 2012.

[Gao *et al.*, 2022] Jiechao Gao, Haoyu Wang, and Haiying Shen. Task failure prediction in cloud data centers using deep learning. *IEEE Transactions on Services Computing*, 15:1411–1422, 2022.

[Ghallab *et al.*, 2004] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004.

[Gonçalves and Silva, 2013] Ivo Gonçalves and Sara Silva. Balancing learning and overfitting in genetic programming with interleaved sampling of training data. In *Proceedings of the European Conference on Genetic Programming (EuroGP)*, pages 73–84. Springer, 2013.

[Hajihosseini *et al.*, 2018] Payman Hajihosseini, Mohammad Mousavi Anzehaee, and Behzad Behnam. Fault detection and isolation in the challenging Tennessee Eastman Process by using image processing techniques. *ISA Transactions*, 79:137–146, 2018.

[Hastie *et al.*, 2009] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

[Havelund and Peled, 2023] Klaus Havelund and Doron Peled. Monitorability for runtime verification. In Panagiotis Katsaros and Laura Nenzi, editors, *Proceedings of the 23rd International Conference on Runtime Verification (RV 2023)*, volume 14245 of *Lecture Notes in Computer Science*, pages 447–460. Springer, 2023.

[Holt *et al.*, 2023] Samuel Holt, Zhaozhi Qian, and Mihaela van der Schaar. Deep generative symbolic regression. *arXiv preprint arXiv:2401.00282*, 2023.

[Huang, 2017] X. Huang. Hard drive failure prediction for large scale storage system. 2017. M.Sc Thesis.

[Kim and Sohn, 2020] Tae San Kim and So Young Sohn. Multitask learning for health condition identification and remaining useful life prediction: Deep convolutional neural network approach. *Journal of Intelligent Manufacturing*, 32(8):2169–2179, 2020.

[Koza, 1994] John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, 1994.

[Kupferman and Vardi, 2001] Orna Kupferman and Moshe Y Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[Leucker and Schallhart, 2009] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[Lu *et al.*, 2020] Sidi Lu, Bing Luo, Tirthak Patel, Yongtao Yao, Devesh Tiwari, and Weisong Shi. Making disk failure predictions SMARTer! In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)*, pages 151–167. USENIX Association, 2020.

[Maler and Nickovic, 2004] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Proceedings of the Joint International Conferences on Formal Modelling and Analysis of Timed Systems (FORMATS 2004) and Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2004)*, pages 152–166. Springer, 2004.

[Mundhenk *et al.*, 2021] Terrell Mundhenk, Mikel Landajuela, Ruben Glatt, Claudio P Santiago, Brenden K Petersen, et al. Symbolic regression via deep reinforcement learning enhanced genetic programming seeding. *Advances in Neural Information Processing Systems*, 34:24912–24923, 2021.

[Ničković and Yamaguchi, 2020] 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 2020)*, volume 12302, pages 564–571. Springer, 2020.

[Onel *et al.*, 2019] Melis Onel, Chris A. Kieslich, and Efstratios 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*, 65(3):992–1005, 2019.

[Petmezas *et al.*, 2021] Georgios Petmezas, Kostas Haris, Leandros Stefanopoulos, Vassilis Kilintzis, Andreas Tzavelis, John A Rogers, Aggelos K Katsaggelos, and Nicos Maglaveras. Automated atrial fibrillation detection using a hybrid CNN-LSTM network on imbalanced ECG datasets. *Biomedical Signal Processing and Control*, 63:102194, 2021.

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

[Qian *et al.*, 2021] Kai Qian, Jie Jiang, Yulong Ding, and Shuang-Hua Yang. DLGEA: A deep learning guided evolutionary algorithm for water contamination source identification. *Neural Computing and Applications*, 33:11889–11903, 2021.

[Williams and Zipser, 1989] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.