RTAMT- Runtime Robustness Monitors with Application to CPS and Robotics

Tomoya Yamaguchi[1], Bardh Hoxha[1] and Dejan Ničković[2]

[1]TRINA, Toyota Motor NA R&D, 1555 Woodridge Ave, Ann Arbor, 48105, Michigan, U.S.
[2]AIT Austrian Institute of Technology, Giefinggasse 4,Vienna, 1210, Vienna, Austria.

Contributing authors: tomoya.yamaguchi@toyota.com; bardh.hoxha@toyota.com; dejan.nickovic@ait.ac.at;

Abstract

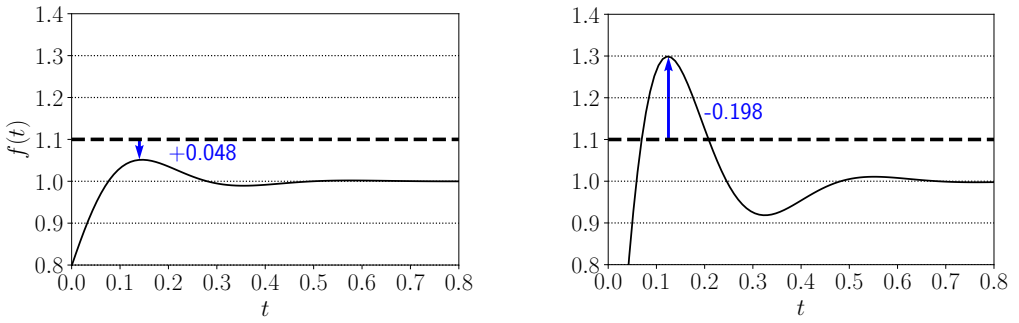
In this paper, we present Real-Time Analog Monitoring Tool (RTAMT), a tool for quantitative monitoring of Signal Temporal Logic (STL) specifications. The library implements a flexible architecture that supports: (1) various environments connected by an Application Programming Interface (API) in Python, (2) various flavors of temporal logic specification and robustness notion such as STL, including an interface-aware variant that distinguishes between input and output variables, and (3) discrete-time and dense-time interpretation of STL with generation of online and offline monitors. We specifically focus on robotics and Cyber-Physical Systems (CPSs) applications, showing how to integrate RTAMT with (1) the Robot Operating System (ROS) and (2) MATLAB/Simulink environments. We evaluate the tool by demonstrating several use scenarios involving service robotic and avionic applications.

Keywords: runtime verification, formal specifications, robotics, cyber-physical systems

1 Introduction

Cyber-Physical Systems (CPSs) [1–3] are systems which integrate both cyber and physical components, operating in complex environments and increasingly featuring autonomous decision-making capabilities enabled by machine learning. Distributed robotic applications are typical examples of autonomous CPS used in contexts ranging from collaborative manufacturing environments to assisted living. Other examples of CPSs include smart buildings that learn their practitioner's profile and accordingly optimize heating strategies, autonomous vehicles that are able to drive without human intervention or medical devices that modify a therapy according to the patient's needs. To address complexity challenges, various frameworks have been proposed to facilitate development. For instance, Robot Operating System (ROS) [4] provides a meta-operating system with tools and libraries to help engineers develop robotic applications while MATLAB/Simulink® enables the modeling of CPS control applications.

Under these circumstances, Verification and Validation (V&V) remains a bottleneck as state-of-the-art techniques do not scale to this level of complexity, making static safety assurance a very costly and time-demanding, if not impossible, activity. Runtime Assurance (RTA) [5], an alternative approach for ensuring the safe operation of robotic and other sophisticated CPSs, is used when static verification is not possible. RTA allows the use of unverified components in a system that implements a safe fallback mechanism for



(a) Satisfied (b) Violated

Fig. 1: Two typical PID controller behaviors evaluated against the specification "it is always the case that $f(t)$ is smaller or equal to 1.1" ($\mathbf{G}\,(f(t) \leq 1.1)$) in STL: apart from the satisfied/violated qualitative verdict, the robustness semantics provides additional quantitative feedback (a) satisfied with $robustness = +0.048$ (b) violated with $robustness = -0.198$.

(1) detecting anomalies during real-time system operations and (2) invoking a recovery mechanism that can bring the system back to its safe (and possibly degraded) operation. Runtime verification provides a reliable, rigorous and systematic way for finding violation in system executions and consequently represents a viable solution for the monitoring part of an RTA entity. Runtime verification and assurance techniques must support seamless integration in common design and operation environments.

Formal specifications play an important role in runtime verification and enable to precisely express intended system properties. Signal Temporal Logic (STL) [6] is a formal specification language that is used to describe CPS requirements. STL can provide quantitative *robustness* semantics, which is used to measure to which extent an observed behavior satisfies or violates a given specification. For instance, Fig. 1 shows two typical PID controller behaviors. The STL specification requires the observed behavior $f(t)$ to always remain below 1.1. The first behavior (Fig. 1 (a)) satisfies its specification with the positive robustness $+0.048$. This value corresponds to the closest distance between $f(t)$ and the 1.1 threshold. In contrast, the second behavior (Fig. 1 (b)) violates that same specification with negative robustness -0.198. This value represents the largest amount that $f(t)$ goes above 1.1. We observe that the real-valued evaluation contrasts the classical satisfied/violated answer that we typically get from reasoning with the qualitative interpretation of specification languages.

To address the above V&V challenges, we present Real-Time Analog Monitoring Tool (RTAMT)[1], a versatile library for generating monitors from STL specifications. The contributions of this work are as follows:

1. We integrate RTAMT into multiple design and operational environments with its Application Programming Interface (API) – the Python library's API allows for this integration. We demonstrate the integration of RTAMT with ROS[2] and MATLAB/Simulink.
2. The tool facilitates temporal logic-based specifications and quantitative/robustness notions such as STL. We allow integration of various syntactic and semantic variants of the language. For instance, the tool supports standard STL and its Interface-Aware extension (IA-STL) [7]. We also provide a tool library to support implementing one's own temporal logic-based specification language.
3. We implement both *offline* and *online* monitors with either *discrete-time* or *dense-time* interpretation of the behaviors and the specification language.

[1] RTAMT:https://github.com/nickovic/rtamt
[2] RTAMT4ROS:https://github.com/nickovic/rtamt4ros

This work is an extended version of the conference paper [8]. We briefly summarize the additional content that is provided in this paper, compared to its conference variant:

- A detailed presentation of the library's architecture.
- An extensive description of how RTAMT can be integrated to ROS and MATLAB/Simulink®.
- A comprehensive evaluation section with additional experiments and case studies.
- More extensive and complete related work section.

The layout of the paper is as follows. First, we present definitions of STL and its extensions and sketch the monitoring procedures in Sec. 2; we then give the RTAMT architecture in Sec. 3, its API in Sec. 4, and its library in Sec. 5. These enable the use of specification-based runtime verification and assurance methods. In Sec. 6, we give our detailed evaluation of the tool and case studies in robotic applications based on ROS and MATLAB/Simulink. We then present related work in Sec. 7, and finally conclude the paper in Sec. 8.

2 Monitoring Temporal Logic Specifications

RTAMT is a tool for the automatic generation of monitors from declarative specifications. Given an input signal in the form of a sequence of (time, value) pairs and a specification, RTAMT computes at different points in time how robust the observed signal is compared to the specification, i.e. how far is it from satisfying or violating it. STL is utilized as the specification language of choice in RTAMT, which supports discrete-time and dense-time monitors for usage in a wide array of applications.

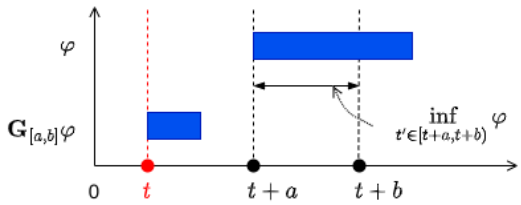
In Sec. 2.1, we provide an overview of the syntax of STL with the standard *future temporal operators* and more specialized *past temporal operators*, which are more suitable for online monitoring requirements. With a specification that includes only past temporal operators, the robustness of a signal with respect to the specification may be evaluated at the current timestep and does not depend on future observations. In addition, we provide semantics that support discrete and dense temporal models. In Sec. 2.2 we present a pastification procedure that enables translation of future temporal operators to past temporal operators. Finally, in Sec. 2.3 we introduce the *interface-aware* extension of STL (IA-STL) that distinguishes between *input* and *output* variables. We use IA-STL to demonstrate how the library can be extended with other specification formalisms such as temporal operators and additional robustness metrics.

2.1 Signal Temporal Logic

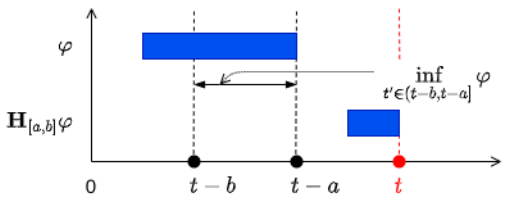
Signal Temporal Logic (STL) extends Linear Temporal Logic (LTL) with real-time temporal operators and numerical predicates defined over real-valued behaviors. We interpret STL over *finite signals* that we represent as finite sequences of (time, value) pairs. Let $X = \{x_1, \ldots, x_n\}$ be a set of real-valued variables. A valuation $v : X \to \mathbb{R}$ For $x \in X$ maps a variable x to a real value. A signal w defined over X is a function $\mathbb{T} \to \mathbb{R}^X$ that gives the value $w(t)$ of the variables in X at time $t \in \mathbb{T}$, where $\mathbb{T}$ is a finite interval $[0, d)$. A signal w can be also seen as a vector of real-valued signals $w_x : \mathbb{T} \to \mathbb{R}$ associated to variables $x \in X$. Given two signals $w^1 : \mathbb{T} \to \mathbb{R}^{n_1}$ and $w^2 : \mathbb{T} \to \mathbb{R}^{n_2}$, we define their composition $w^1 \,||\, w^2$ as the function $\mathbb{T} \to \mathbb{R}^{n_1+n_2}$, with the expected meaning. Given a subset $Y \subseteq X$ of variables, we define the projection w_Y of the signal w to Y as the composition $||_{y \in Y}\, w_y$.

We can have two interpretations of signals – discrete- and dense-time. For the discrete-time interpretation of signals, we assume that the signal is periodically sampled with some period Δ. It follows that every sample can be uniquely indexed with an integer i corresponding to the sampling time $i\Delta$. Consequently, the time domain $\mathbb{T} = [0, d]$ is an interval of integers with $d \in \mathbb{N}$. For the dense-time interpretation, we assume that the time domain $\mathbb{T} = [0, d]$ is an interval of reals with $d \in \mathbb{Q}_{>0}$. In contrast to the discrete-time interpretation, the changes in the dense-time signal can happen anywhere in the interval $[0, d]$. While we assume signals with finite variability, we do not impose a bound on the number of changes in any unit time.

Both discrete-time and dense-time signals can be represented with a finite sequence $(t_0, v_0), \ldots, (t_n, v_n)$ of (timestamp, valuation)



(a) $\rho(\mathbf{G}\,[a,b]\,\varphi, w, t) = \inf_{t' \in [t+a, t+b)} \rho(\varphi, w, t')$



(b) $\rho(\mathbf{H}\,_{[a,b]}\,\varphi, w, t) = \inf_{t' \in (t-a, t-b]} \rho(\varphi, w, t')$

Fig. 2: Semantics for the future timed operator **G** (globally) and the past timed operator **H** (historically): Input φ is described as Booleans for simplicity. The difference is (a) $t' \in [t+a, t+b]$ in $\mathbf{G}_{[a,b]}$ refers to future points in time with respect to the current time t, In contrast, (b) $t' \in [t-a, t-b]$ in $\mathbf{H}_{[a,b]}$ refers to past points in time with respect to the current time t,

pairs such that $t_0 = 0$ and $\forall i \in [1, n] : t_i \in \mathbb{T}$ and $t_{i-1} < t_i$, v_i is a valuation over X and $t_n = d$. For the dense-time interpretation of the specification language, we assume that signals are piecewise-constant, i.e. for all $t \in [t_i, t_{i+1})$ and $x \in X$, $w(x,t) = w(x,t_i)$.

The basic building block in STL is a *predicate* $f(Y) > c$, where $f(Y)$ is a term with $Y \subseteq X$, $f : \mathbb{R}^Y \to \mathbb{R}$ is an interpreted function and c is a real number. The syntax of an STL formula φ is defined recursively with the following grammar,

Definition 1 (Syntax of STL)

$$\varphi := f(Y) > c \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1\, \mathbf{U}_I \varphi_2 \mid \varphi_1\, \mathbf{S}_I \varphi_2$$

where $Y \subseteq X$, I is an interval of the form $[a,b]$ or $[a,\infty)$ where $a \leq b$ are rational numbers.

The operators for disjunction ($\vee$) and negation ($\neg$) are defined in the usual way. **U** (until) and **S** (since) are temporal operators. The syntax of STL is very similar to that of LTL, with the addition of numerical predicates of the form $f(Y) > c$ and intervals I that bound the scope of the temporal operators.

Given an STL formula φ, a signal w and a time $t \in \mathbb{T}$, we define the *quantitative* (or *robustness*) semantics $\rho(\varphi, w, t)$ as follows:

Definition 2 (Quantitative Semantics of STL)

$$\begin{aligned}
\rho(f(Y) > c, w, t) &= f(w_Y(t)) - c \\
\rho(\neg\varphi, w, t) &= -\rho(\varphi, w, t) \\
\rho(\varphi_1 \vee \varphi_2, w, t) &= \max\left(\rho(\varphi_1, w, t), \rho(\varphi_2, w, t)\right) \\
\rho(\varphi_1\, \mathbf{U}_I \varphi_2, w, t) &= \\
&\sup_{t' \in t \oplus I \cap \mathbb{T}} \min\left(\rho(\varphi_2, w, t'), \inf_{t'' \in [t, t')} \rho(\varphi_1, w, t'')\right) \\
\rho(\varphi_1\, \mathbf{S}_I \varphi_2, w, t) &= \\
&\sup_{t' \in t' \ominus I \cap \mathbb{T}} \min\left(\rho(\varphi_2, w, t'), \inf_{t'' \in (t', t]} \rho(\varphi_1, w, t'')\right)
\end{aligned}$$

where $\min(P)$ and $\max(P)$ denote the smallest and the largest elements in the set P, while $\inf(P)$ and $\sup(P)$ denote the infinum (greatest lower bound) and the supremum (least upper bound) of P.

The quantitative semantics from Definition 2, intuitively measures how much should the signal be modified in order to satisfy or violate the specification. The measured value is also known as *spatial robustness*. It can be seen as (an approximation of) the distance between the observed behavior and the boundary of the set representing all behaviors that satisfy the property.

We observe that the **U** (Until) and **S** (Since) operators may be utilized to derive other temporal operators:

Future Temporal Operators

$$\begin{aligned}
&\text{finally} && \mathbf{F}_I \varphi &&\equiv \mathsf{true}\, \mathbf{U}_I \varphi \\
&\text{globally} && \mathbf{G}_I \varphi &&\equiv \neg \mathbf{F}_I \neg\varphi \\
&\text{next} && \mathbf{X}\, \varphi &&\equiv \mathsf{false}\, \mathbf{U}_{[0,\infty)} \varphi
\end{aligned}$$

Past Temporal Operators

$$\begin{aligned}
&\text{once} && \mathbf{O}_I \varphi &&\equiv \mathsf{true}\, \mathbf{S}_I \varphi \\
&\text{historically} && \mathbf{H}_I \varphi &&\equiv \neg \mathbf{O}_I \neg\varphi \\
&\text{previous} && \mathbf{Y}\, \varphi &&\equiv \mathsf{false}\, \mathbf{S}_{[0,\infty)} \varphi \\
&\text{rise} && \uparrow \varphi &&\equiv \mathbf{Y} \neg\varphi \wedge \varphi \\
&\text{fall} && \downarrow \varphi &&\equiv \mathbf{Y}\, \varphi \wedge \neg\varphi
\end{aligned}$$

Intuitively, a signal $w_Y(t)$ satisfies a formula $\varphi_1 \mathbf{U}_{[a,b]} \varphi_2$ at time t if there exists a time $t' \in [t+a, t+b]$ such that w satisfies φ_2 and for all the times before then w satisfies φ_1. Since the robustness evaluation needs to consider the signal at future timesteps, $\mathbf{U}$ and derived operators $\mathbf{F}$, $\mathbf{G}$, $\mathbf{X}$ are categorized as future operators. Similarly, a signal $w_Y(t)$ satisfies a formula $\varphi_1 \mathbf{S}_{[a,b]} \varphi_2$ at time t if there exists a time $t' \in [t-a, t-b]$ such that w satisfies φ_2 and for all the times after then w satisfies φ_1. $\mathbf{S}$ and derived operators $\mathbf{O}$, $\mathbf{H}$, $\mathbf{Y}$ are referred to as past operators. The intuition behind these operators is presented visually in Figs. 2a and 2b. We note that discrete-time quantitative semantics of STL are evaluated only at sampled instances. Past temporal operators are more suitable for monitoring applications than future temporal operators because robustness evaluation relies only on current and past time valuations of a signal.

We note that Definition 2 can be used for both the discrete and dense time interpretation of the logic, depending on whether t, t' and t'' are quantified over naturals or reals. We also observe that the next $\mathbf{X}$, previous $\mathbf{Y}$, rise $\uparrow$ and fall $\downarrow$ operators are only meaningful for the discrete-time interpretation of STL.

In the following, we consider two subsets of STL:

- *Bounded-Future* Signal Temporal Logic (bfSTL) where the unbounded time intervals are not permitted. That is, the $\mathbf{U}_I$ operator has a defined interval $[a, b]$ where a and b are in $\mathbb{R}^+$.
- *Past* Signal Temporal Logic (pSTL) that only supports past-operators.

To facilitate the online monitoring of bounded future properties in RTAMT, we use a *pastification* procedure, which takes a bfSTL formula and generates an equi-satisfiable pSTL formula for monitoring purposes.

2.2 Pastification from bfSTL to pSTL

Monitoring specifications with future temporal operators is challenging because the evaluation at time index t may depend on the observed inputs at some future time indices $t' > t$. bfSTL specifications have a bounded future horizon h that can be syntactically computed from the formula structure. For such specifications, the online monitoring challenge can be addressed by postponing the formula evaluation from time index t to the end of the (bounded) horizon $t+h$, where all the inputs necessary for computing the robustness degree are available. In this section, we briefly sketch this procedure, called *pastification* [9, 10].

We first define the *temporal depth* $H(\varphi)$ of φ as the syntax-dependent upper bound on the actual depth of the specification. It corresponds to the maximum time in the future that is relevant for the evaluation of the specification now, which is inductively computed as follows[3]:

Definition 3 (Temporal Depth)

$$
\begin{aligned}
H(f(Y) > c) &= 0 \\
H(\neg\varphi) &= H(\varphi) \\
H(\varphi_1 \vee \varphi_2) &= \max\{H(\varphi_1), H(\varphi_2)\} \\
H(\mathbf{X}\,\varphi) &= H(\varphi) + 1 \\
H(\varphi_1 \mathbf{U}_{[a,b]} \varphi_2) &= b + \max\{H(\varphi_1), H(\varphi_2)\}
\end{aligned}
$$

In the next step, we define the pastification operation Π on the *STL* formula φ with past and bounded future and its bounded horizon $d = H(\varphi)$. To enable this transformation of specifications, we define a new *precedes* $\mathbf{P}_{[a,b]}$ auxiliary temporal operator, which is essentially implementing the bounded until operator interpreted from the end of the bounded formula horizon:

$$
\rho(\varphi_1 \mathbf{P}_{[a,b]} \varphi_2, w, t) = \sup_{t' \in [t-b+a, t]} \min \left(\rho(\varphi_2, w, t'), \inf_{t'' \in (t-b, t']} \rho(\varphi_1, w, t'') \right)
$$

The pastification procedure essentially takes a bounded future formula φ with horizon $H(\varphi)$ and gives the recipe how to shift the evaluation of that formula from time t to time $t + H(\varphi)$, when all the information required for the evaluation of the formula becomes available.

[3] The pastification of the $\mathbf{X}$ operator is relevant only for the discrete-time interpretation of the specification language.

Definition 4 (Pastification Operation)

$$
\begin{aligned}
\Pi(f(Y) > c, d) &= \mathbf{O}_{[d,d]}(f(Y) > c) \\
\Pi(\neg\varphi, d) &= \neg\Pi(\varphi, d) \\
\Pi(\varphi_1 \vee \varphi_2, d) &= \Pi(\varphi_1, d) \vee \Pi(\varphi_2, d) \\
\Pi(\mathbf{X}\,\varphi, d) &= \Pi(\varphi, d-1) \\
\Pi(\mathbf{F}_{[a,b]}\varphi, d) &= \mathbf{O}_{[0,b-a]}\Pi(\varphi, d-b) \\
\Pi(\varphi_1\,\mathbf{U}_{[a,b]}\varphi_2, d) &\leftrightarrow \Pi(\varphi_1, d-b)\mathbf{P}_{[a,b]}\Pi(\varphi_2, d-b)
\end{aligned}
$$

Formally, we say that for an arbitrary bfSTL formula φ, signal w and time index $t \in \mathbb{N}$, $\rho(\varphi, w, t) = \rho(\Pi(\varphi), w, h(\varphi))$.

Example 1 The pastification of the bfSTL specification $\varphi \equiv (req \geq 3) \rightarrow \mathbf{F}_{[0,5]}(gnt \geq 3)$ from the running example corresponds to the pSTLformula $\Pi(\varphi) \equiv \mathbf{O}_{[5,5]}(req \geq 3) \rightarrow \mathbf{O}_{[0,5]}(gnt \geq 3)$.

2.3 Interface-Aware STL

Interface-aware Signal Temporal Logic (IA-STL) extends STL by classifying variables appearing in the specification as *input* or *output* variables [7]. This simple addition to the specification language is fundamental to reasoning about *open* systems and allows their specification as input/output relations, rather than sets of correct execution traces. IA-STL allows questions that cannot be formulated with the general STL formulas:

- How good is the reaction of the system to a given input signal with respect to its requirements?
- Does a concrete input signal exercise the system in any meaningful way with respect to its requirements?

IA-STL admits several semantic interpretations, one for each question that the interpretation shall answer. There are two particularly useful IA-STL semantics:

Output robustness measures how robust a specification is relative to the set output signals. The real-valued (positive or negative) robustness value (positive or negative) indicates how much the signal can be perturbed and still satisfy/violate the specification. When (plus or minus) infinity, the output robustness indicates that the specification is vacuously satisfied or not satisfied by the given input signal.
Input vacuity represents the level of vacuity of the specification with respect to a given input signal. When a positive or negative real, it indicates how much one can change the input without violating or satisfying the specification. When equal to 0, the input vacuity indicates the specification is non-vacuously exercised by the input signal. Intuitively, it means that the specification is not robust to even a slightest change in the inputs because it might induce an output behavior that affects the robustness in an arbitrary manner.

We illustrate these two notions of robustness in Fig. 3, which depicts four simple request-grant behaviors that we evaluate against the IA-STL bounded response property $\mathbf{G}\,(req \geq 3 \rightarrow \mathbf{F}_{[0,5]}gnt \geq 3)$. The first behavior (Fig. 3a) satisfies the specification with output robustness $+3$ because the *gnt* signal is 3 units away from the threshold in the interval of interest. Since the specification is not vacuously satisfied by the input *req*, the input vacuity equals to 0. The second behavior (Fig. 3b) satisfies the specification with robustness $+1$, but with output robustness $+\infty$ because the request is never issued and hence the specification is vacuously satisfied. The input vacuity also equals to $+1$ because changing the values of the input *req* by any amount smaller than 1 would still guarantee vacuous satisfaction of the specification. The third behavior (Fig. 3c) violates the specification with output robustness -2, corresponding to the distance of *gnt* from the threshold. Finally, the fourth behavior (Fig. 3d) also violates the specification with robustness -1, but with output robustness -2. The classical robustness corresponds to the distance between the input *req* and the threshold – intuitively, the least expensive way to achieve the satisfaction of the specification is to decrease the amplitude of the input until the specification is vacuously satisfied. However, when measuring output robustness, the input is fixed and cannot change, thus yielding the value of -2, which corresponds to the distance between the output *gnt* and the threshold. To enable output robustness and input vacuity, we first lift the notion of robustness ρ, to the more general notion of U-robustness relative to V, denoted by ρ^V_U, where $U \subseteq Y \subseteq X$ and $V \subseteq Y \subseteq X$, and $U \cap V = \emptyset$. We define the robust semantics $\rho^V_U(\varphi, w, t)$ by induction, where the only difference from the definition of $\rho(\varphi, w, t)$ is the case of

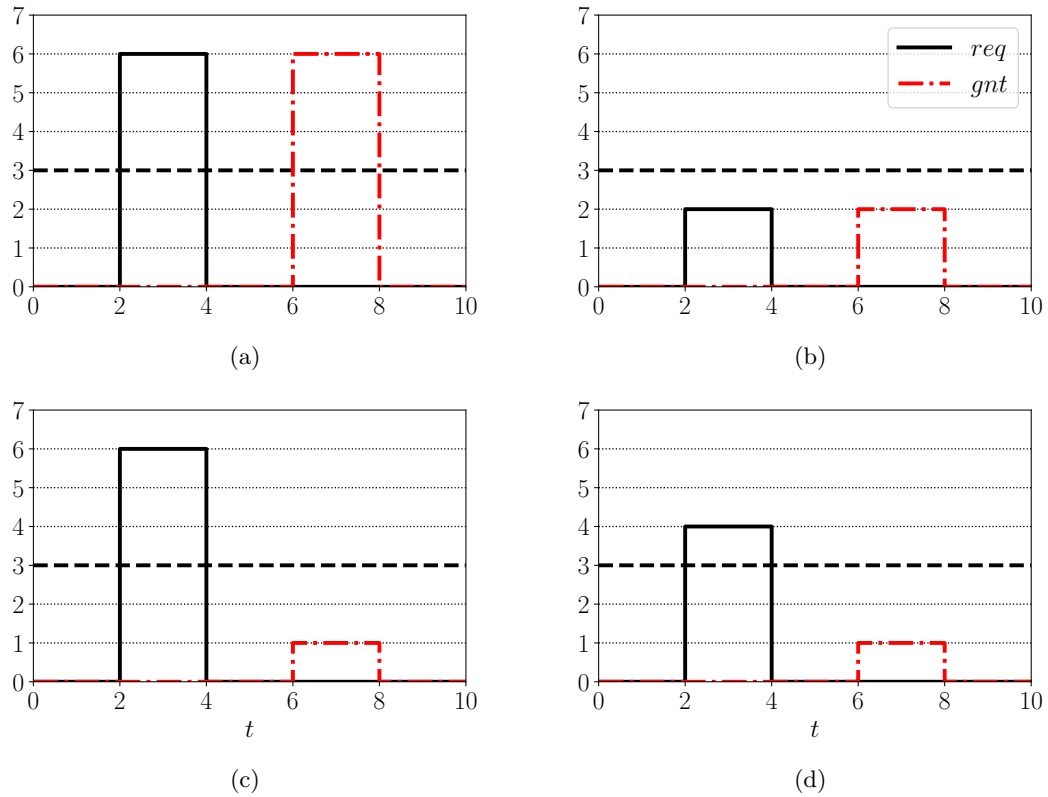


Fig. 3: Examples of IA-STL evaluating request-grant signals with the formula $\mathbf{G}\,(req \geq 3 \rightarrow \mathbf{F}_{\,[0,5]} gnt \geq 3)$: A request is issued whenever the value of the signal req is greater or equal to 3. Similarly, the grant is issued whenever the value of the signal gnt is greater or equal to 3. The specification requires that every request is eventually followed within 5 time units by a grant. The sub-figures illustrate 4 different behaviors where the specification is (a) satisfied with with robustness 3, output robustness +3, and input vacuity 0, (b) vacuously satisfied with that (the request is never issued) with robustness +1, output robustness $+\infty$, and input vacuity +1, (c) does not satisfy that with robustness -2 and output robustness -2, (d) also does not satisfy that with robustness -1 and output robustness -2.

numeric predicates:

$$\rho^V_U(f(Y) > c, w, t) = \begin{cases} 0 & \text{if } Y \not\subseteq U \cup V \\ fw_Y(t)) - c & \text{else if } Y \not\subseteq V \\ \text{sign}(f(w_Y(t)) - c) \cdot \infty & \text{otherwise} \end{cases}$$

where for all $a \in \mathbb{R}$, $\text{sign}(a) \cdot \infty = +\infty$ if $a > 0$, $-\infty$ otherwise. Intuitively, $\rho(\varphi, w, t)$ measures the robustness of a specification to the signals in U relative to the signals in V. Hence, we use the classical robustness computation to compute predicates over variables in U, but we treat predicates over variables in V and $Y \backslash (U \cup V)$ differently. The predicates over variables in V are given qualitative evaluation, resulting in robustness that can be either $+\infty$ or $-\infty$. The rationale is that we consider signals in V to be *fixed* and that a change resulting in switching the satisfaction or the violation of the predicates results in an infinite cost. On the other hand, we consider signals in $Y \backslash (U \cup V)$ to be *uncontrollable* and that even an infinitesimally small change could result in switching the satisfaction status of the predicate.

We are now ready to define the syntax and semantics of IA-STL.

Definition 5 (Interface-Aware Signal Temporal Logic) An Interface-Aware Signal Temporal Logic

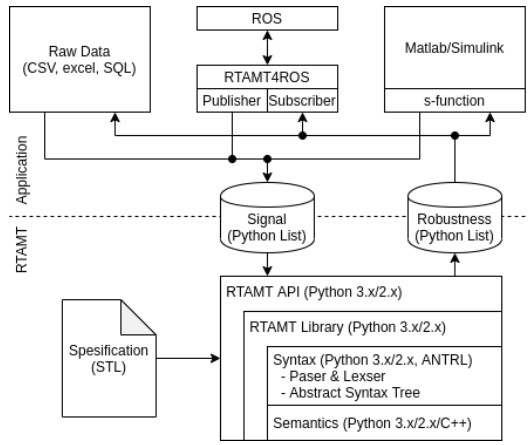


Fig. 4: Overview of RTAMT.

(IA-STL) specification over X is a tuple (X_U, X_V, φ), where $X_U, X_V \subseteq X$, $X_U \cap X_V = \emptyset$ and φ is a STL formula.

We define *output robustness* and *input vacuity* using the notion of relative robustness:

Output robustness that is denoted μ, as the X_V-robustness relative to $X \backslash X_V$.
Input vacuity that is denoted ν, the X_U-robustness relative to $\emptyset$.

3 Architecture

In this section, we introduce the overview of the RTAMT architecture for specification-based monitoring. Fig. 4 depicts the overview of RTAMT. The core module of RTAMT API is developed in Python[4]. for several reasons:

1. Facilitates handling common input data formats such as CSV, Excel, or SQL server.
2. Easy integration with ROS, a well-known middleware in the robotics domain.
3. Better connectivity with MATLAB/Simulink, a well-known Model Based Development (MBD) platform in the control design domain.
4. Sophisticated software package system: Python Package Index (PyPI) enables smooth distribution.
5. Popularity and thereby increasing user base.

[4]Both Python 2.x and 3.x are supported for ROS and general purposes.

For practitioners, we provide a flexible and modular library behind the API that enables them to implement their own specification language based on LTL/STL. Fig. 5 shows the architecture of the RTAMT Library in a class diagram. The cores of the library are the syntax layer, which manages specification with lexer, parser, and Abstract Syntax Tree (AST), and the semantic layer, which calculates robustness from given trajectories with specific semantics. Since Python is an interpreted language, there is a natural trade-off between computing speed and ease of programming. For this reason, we implement the syntax layer in Python, because the layers are executed only one time in the parse phase, and we prioritize flexibility rather than computing efficiency. In contrast to the syntax layer, the semantic layer is called upon every monitoring update. Because we expect real-time monitoring, we can exploit not only Python for rapid prototyping, but also C++ for fast computing in this layer. Next we explain each layer in detail.

3.1 Syntax Layer

We first utilize ANTLR4 [11] lexer and parser to parse a given specification. ANTLR4 provides a flexible, easy-to-use platform for lexer and parser instantiated with a tool-specific setting file ANTLR4 grammar. `StlAst` and `StlAstVisitor` are instances of the RTAMT AST and visitor class in STL case as the core of this layer and the link to the next semantics layer.

Another functionality in this layer is handling any syntactic manipulations of the specification parse trees, including the pastification procedure from Sec. 2.2. In that case, we translate the bfSTL formula ϕ into an equi-satisfiable pSTL formula ψ, which uses only past temporal operators. The actual pastification parser `StlPastifier` is extended from `StlAstVisitor`, which provides the parse method for translating the specification text into an internal representation. This pastification two-step process is depicted in Fig. 6.

3.2 Semantics Layer

This semantics layer provides algorithms for online and offline monitors from declarative specifications with quantitative semantics that are inductively implemented by traversing the AST from the syntax layer.

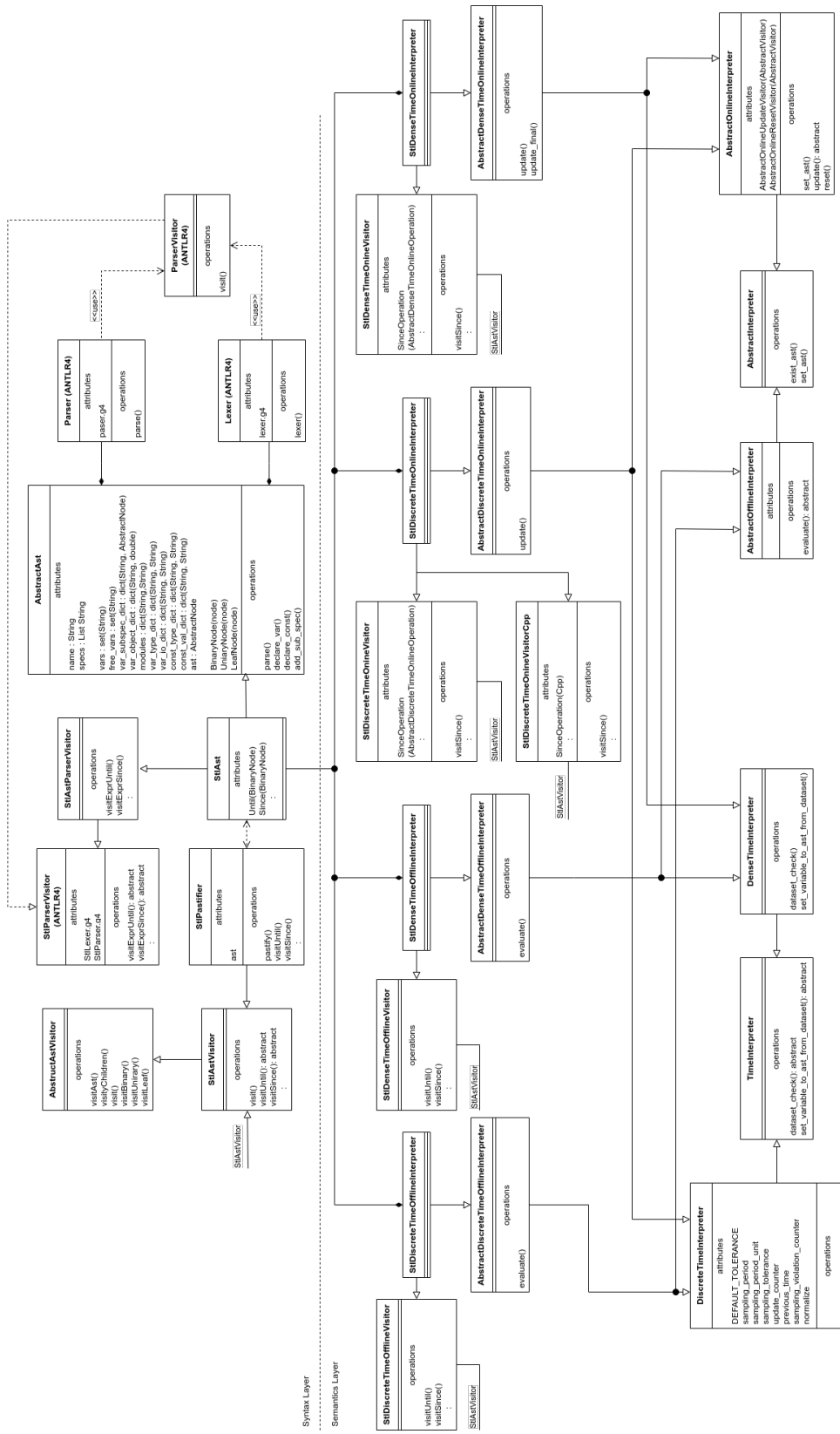


Fig. 5: Architecture of RTAMT.

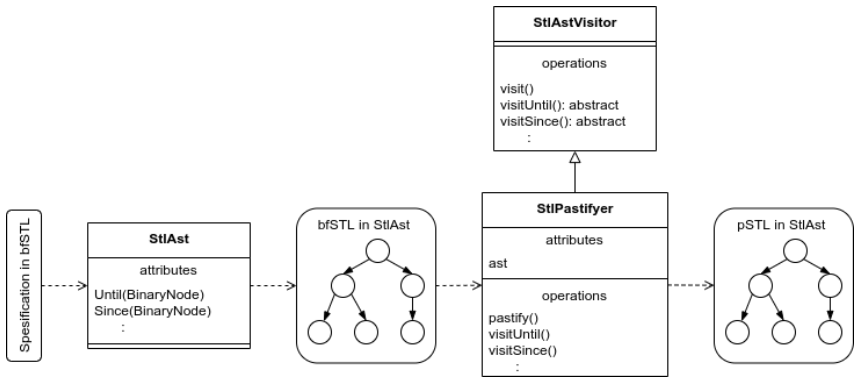


Fig. 6: Pastification from bfSTL to pSTL.

The implementation of the monitoring algorithm is specific to the specification language used. The time handler (`TimeInterpreter`) handles two signal classes.

Discrete-time (`DiscreteTimeInterpreter`) This class is the base for monitoring discrete-time signals. The implementation follows a time-triggered approach in which sensing of inputs and output generation are done at a periodic rate. Its use is motivated by [12], which shows that by weakening/strengthening real-time specifications, discrete-time evaluation of properties preserves important properties of dense-time interpretation. This approach admits an upper bound on the use of computation resources. The discrete-time monitors essentially implement the algorithm from [10] adapted to the robustness semantics.
Dense-time (`DenseTimeInterpreter`) This class is the base for monitoring dense-time signals, which use piece-wise constant interpolation. The dense-time monitors implement the algorithms from [13] and adapt them to the piece-wise constant interpretation of signals. The resulting monitors follow an event-driven approach, where the samples can happen at any time in an intervals of reals and there can be an arbitrary (but finite) number of samples in any unit interval of time. This approach is suitable for more advanced and distributed applications in which state variable observations are not periodically triggered but rather driven by external, hence uncontrollable events. The key algorithmic ingredient in the offline evaluation of STL interpreted over dense-time signals is an optimal streaming algorithm for computing the minimum (or the maximum) values of numeric sequences (samples) over a sliding window of fixed size. This procedure allows to compute efficiently temporal operators $\mathbf{G}_I$, $\mathbf{F}_I$, $\mathbf{H}_I$ and $\mathbf{O}_I$, whose semantics is defined as a computation of the minimum/maximum values over a sliding window I. It is also adapted to enable the evaluation of the more general $\mathbf{U}_I$ and $\mathbf{S}_I$ operators. To address the online monitoring problem, we use the incremental application of the offline evaluation approach from [14] to the partially received inputs.

In addition to the above, we support two monitoring modes.

Offline Monitor (`AbstractOfflineInterpreter`) This class supports typical offline evaluation, which expects all data to exist at once when evaluated. This provides a method `evaluate` to evaluate the signal.
Online Monitor (`AbstractOnlineInterpreter`) This class supports online evaluation of signals with different timings. The operation thus needs to manage memory to evacuate not incomplete signals until the next signal update. This provides a method `update` to evaluate signals.

Both `evaluate` and `update` method take a list of variable names, `(time, value)` pairs and return a float number representing the robustness degree of the formula at that time index relative to the input prefix observed in the present.

This layer supports a C++ implementation to enable real-time as well. We use the Boost Python library, a C++ library which enables seamless operability between the two languages, to integrate the Python front-end with the C++ back-end.

3.3 Integration of RTAMT to ROS

ROS is the de-facto standard in developing robotic applications, supporting several messaging approaches. In this paper, we assume that ROS nodes use the subscriber/publisher messaging pattern. A publisher categorizes a message into a class (called a topic in ROS) and sends it without any knowledge of who will read the message. A subscriber expresses interest in receiving message from one or more topics and only receives messages of interest, without knowing who sent the message[5]. We also assume that messages are published at a periodic (and known) rate.

The integration of RTAMT into ROS is done in `rospy`, as illustrated in Fig. 7. We assume that we are monitoring a ROS system consisting of one or more nodes that publish messages via variables appearing in the bfSTL specification. The monitoring ROS node uses RTAMT to parse and evaluate the bfSTL formula. In order to communicate with other ROS nodes that publish the relevant messages, the specification must associate variable names with their associated ROS topic names. This is done using annotations. The specification defines which topics the monitor needs to subscribe and publish (topics `rtamt/req`, `rtamt/gnt`, and `rtamt/out` in our example). Since the names of the variables and associated topics as well as the variable types are not known in advance but are inferred from the specification definition, we implemented a dynamic subscribing and publishing mechanism using Python introspection and reflection. For each variable that appears in the specification, we check whether it has a valid ROS message type, generate an instance of the object and retrieve its class information. This information provides a dynamic subscriber and publisher. In addition, a RTAMT monitor is called by a single dynamic callback function as a ROS node and publishes the robustness with the new incoming data from the subscriber.

An existing example of STL evaluation in ROS is shown in Sec. 4.3, and an experiment is shown in Sec. 6.2.

[5]Unless the publisher encodes its identity into the message itself.

3.4 Integration of RTAMT to MATLAB/Simulink

We integrate RTAMT into the MATLAB/Simulink environment via S-functions (short for system functions). S-functions provide a powerful mechanism to extend the existing capabilities of the Simulink environment by enabling to program custom Simulink blocks. S-functions use a special API that allows interaction with the simulation engine similar to how the built-in Simulink blocks interact with the engine. More specifically, we use MATLAB Level 2 S-functions that allow us to integrate the RTAMT API using the MATLAB m-scripts.

An S-function consists of a set of callback methods that are customized to provide the desired functionality. The Simulink engine calls the appropriate method at each stage of the simulation. The main S-function callback methods are associated with the following tasks:

Compilation The stage in which the MATLAB/Simulink engine initializes the S-function and its parameters.
Calculation of outputs at this state, the engine computes the outputs of the block until all its output ports are valid for the current time step.
Update discrete states The block performs activities that are executed only once per time step such as updating discrete states.
Initialize and Terminate Methods Activities required by S-function only once that are performed at the beginning (setting up practitioner's data, initializing state vectors, etc.) and at the end (memory deallocation, etc.) of the simulation.
Integration Computation related to the continuous states and/or nonsampled zero crossings at minor time steps.

The integration of MATLAB/Simulink consists of the following steps:

- We define the RTAMT online monitor S-function block with a single input port consisting of an array of double signals (one for each variable appearing in the specification), a single output signal (the robustness signal) and a single block parameter (the IA-STL specification).
- In the initialize method of the S-function, we create the RTAMT STL specification object,

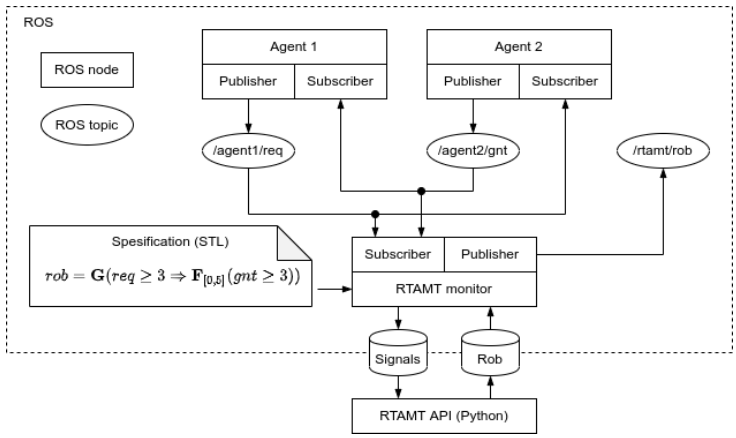


Fig. 7: Integration of RTAMT to ROS.

parse the specification passed as a parameter to the block and pastify the monitor.

- In the method for calculating output, we fetch the most recent array of input data from the input port of the post, pass it to the monitor update function, and forward it to the robustness output port.

An existing example of STL evaluation on MATLAB/Simulink is shown in Sec. 4.5, and an experiment is shown in Sec. 6.3.

4 Application Programming Interface

In this section we show a basic-use case of the RTAMT API in Python with time-stamped data and evaluate it with an arbitrary specification. The three main steps in defining a specification are as follows:

1. define: method for accessing a practitioner-defined specification from a file,
2. parse: method for parsing the specification and building its internal representation,
3. evaluate: method for passing the next snapshot of the input variables and computing the resulting robustness.

4.1 Evaluating STL in Offline

```
1 import rtamt
2
3 spec = rtamt.StlDenseTimeOfflineSpecification()
4 spec.declare_var('req', 'float')
5 spec.declare_var('gnt', 'float')
6 spec.spec = 'G((req>=3)->(F[0,5](gnt>=3)))'
7 spec.parse()
8
9 req = [[0.0, 0.0], [2.0, 6.0], [4.0, 0.0],
      [10.0, 0.0]]
10 gnt = [[0.0, 0.0], [6.0, 6.0], [8.0, 0.0],
      [10.0, 0.0]]
11
12 rob = spec.evaluate(['req', req], ['gnt', gnt])
```

Listing 1: Evaluating STL in offline.

Listing 1 illustrates an example code of an offline monitor. It follows the above steps.

4.2 Evaluating STL in Online

```
1 import rtamt
2
3 spec = rtamt.StlDenseTimeOnlineSpecification()
4 spec.declare_var('req', 'float')
5 spec.declare_var('gnt', 'float')
6 spec.spec = 'H[0,10]((O[5,5](req>=3))->(O[0,5](
      gnt>=3)))'
7 spec.parse()
8
9 req_0 = [[0.0, 0.0], [2.0, 6.0]]
10 gnt_0 = [[0.0, 0.0], [5.0, 6.0]]
11 req_1 = [[4.0, 0.0], [10.0, 0.0]]
12 gnt_1 = [[8.0, 0.0], [10.0, 0.0]]
13
14 rob = spec.update(['req', req_0], ['gnt', gnt_0
      ])
15 rob = spec.update(['req', req_1], ['gnt', gnt_1
      ])
```

Listing 2: Evaluating STL in online.

Listing 2 illustrates an example of online monitor code. Major differences from offline use are that only pSTL is allowed for the specification and the evaluation is done by periodic `update()`. We note that pastification can convert bfSTL to pSTL if we insert `spec.pastify()` after line 7 `spec.parse()`.

4.3 Evaluating STL on ROS

We sketch the ROS interface to RTAMTin RTAMT4ROS.

```
1  def monitor(period, unit):
2      # Init STL SPEC
3      spec = init_spec(period, unit)
4      # Init ROS node
5      rospy.init_node(spec.name, anonymous=True)
6      # Init subscriber and publisher
7      pub = sub_and_pub(spec)
8      # Set monitoring frequency
9      rate = rospy.Rate(spec.
       get_sampling_frequency())
10     # Control loop
11     time_index = 0
12     while not rospy.is_shutdown():
13         rob_msg = mon_update(spec, time_index)
14         pub.publish(rob_msg)
15         time_index += 1
16
17         # Wait until next evaluation
18         rate.sleep()
```

Listing 3: Skeleton of the RTAMT4ROS interface.

The skeleton of the interface, defined by the *monitor* procedure, is shown in Listing 3. The first step consists in initializing the STLspecification that is to be monitored (line 3). The second step consists in initializing the ROS node that hosts the STL monitor (line 5), subscriber, and publisher (line 7), then sets the monitoring frequency (line 9). The STL monitoring node than enters the infinite monitoring loop, which is periodically invoked at the specified frequency (line 18). Each iteration of the loop consists of three steps:

1. automatic invocation of the callback mechanism (Listing 4),
2. invoking the monitoring update (line 13, Listing 7), and
3. publishing the robustness to the appropriate topic (line 14).

```
1  def callback(data, args):
2      spec = args[0]
3      var_name = args[1]
4
5      var = copy.deepcopy(data)
6      spec.var_object_dict[var_name] = var
```

Listing 4: Callback procedure.

The callback procedure, shown in Listing 4, simply receives a new input sample message and copies it to the monitor.

```
1  def init_spec(period, unit):
2      spec = rtamt.
       StlDiscreteTimeOnlineSpecificationCpp()
3      spec.set_sampling_period(period, unit)
4
5      spec.name = 'HandMadeMonitor'
6      spec.import_module('rtamt_msgs.msg', '
       FloatStamped')
7      spec.declare_var('req', 'FloatStamped')
8      spec.declare_var('gnt', 'FloatStamped')
9      spec.declare_var('rob', 'FloatStamped')
10     spec.set_var_topic('req', 'rtamt/req')
11     spec.set_var_topic('gnt', 'rtamt/gnt')
12     spec.set_var_topic('rob', 'rtamt/gnt')
13     spec.spec = \
14         'rob.value = G[0,10]((req.value>=3)->(F
       [0,5](gnt.value>=3)))'
15
16     try:
17         spec.parse()
18         spec.pastify()
19     except rtamt.STLParseException as err:
20         sys.exit()
21     return spec
```

Listing 5: Specification initialization procedure.

In the initialization procedure of specification (see Listing 5), the practitioner creates an STL specification monitor (line 2), sets the sampling period of the monitors (line 3), imports the ROS message type that will be processed by the monitor (line 6), defines the specification variables and the property to monitor (lines 7-12), and parses the specification (line 17). The pastification is applied because the property is bfSTL(line 18).

```
1  def sub_and_pub(spec):
2      # Publish STL output var (robustness)
3      topic = spec.var_topic_dict[spec.out_var]
4      out = spec.get_value(spec.out_var)
5      pub = rospy.Publisher(topic, out.__class__,
       queue_size=10)
6
7      # Subscribe STL input vars
8      for var_name in spec.free_vars:
9          var_object = spec.get_value(var_name)
10         topic = spec.var_topic_dict[var_name]
11         rospy.Subscriber(topic, var_object.
       __class__, callback, [spec, var_name])
12
13     return pub
```

Listing 6: Subscription and publication to ROS topics.

The procedure shown in Listing 6 registers the output variable of the monitor to a publisher, and all the variables appearing in the STL specification to a subscriber. We use reflection and introspection to dynamically determine the variables that need to be registered, and their types.

```
1  def mon_update(spec, time_index):
2      var_name_object_list = []
3      for var_name in spec.free_vars:
4          var_name_object = (var_name, spec.
       get_value(var_name))
5          var_name_object_list.append(
       var_name_object)
6
7      # Update the monitor
8      # spec.update is of the form
9      # spec.update(time_index,[('a',aObj),...)])
```

```
    rob_msg = spec.update(time_index,
     var_name_object_list)

    rob_msg.header.seq = time_index
    rob_msg.header.stamp = rospy.Time.now()

    return rob_msg
```

Listing 7: Monitoring update.

Finally, the monitor updates its evaluation and computes robustness, as shown in Listing 7.

4.4 Textual Specifications for ROS

```
// name
specification spec_x

// imports
from rtamt_msgs.msg import FloatStamped

// variable declarations
input FloatStamped req
output FloatStamped gnt
output FloatStamped rob

// annotations
@ topic(req, rtamt/req)
@ topic(gnt, rtamt/gnt)
@ topic(rob, rtamt/rob)

// bfSTL property
rob.value = G[0,10]((req.value>=3)->(F[0,5](gnt.
    value>=3)))
```

Listing 8: The textual form "spec.stl".

The tool supports an automatic monitor setting for ROS with a textual form for non-programmer practitioners. The concrete textual specifications consist of: the name header, a list of Python modules to be imported, a list of variable declarations, a list of annotations, and the property. Listing 8 formulates this STL property. The syntax of the specification language allows the use of arbitrary data types. We assume that variables `req` and `gnt` are defined as objects of Python type `FloatStamped` with attributes `header` of type `Header` and `value` of type `float` [6]. We also need to import the module that defines the arbitrary data type used in the specification (see line 5). The annotations are special comments that are by default ignored but that can provide additional information in specific contexts. For now we ignore the annotations from lines 13-15. We finally note that the specification[7] robustness is assigned to one of the declared variables (`rob.value` in line 18). Practitioners can launch these textual specifications in ROS below,

```
$ rosrun rtamt4ros ros_stl_monitor.py
↪  spec.stl --period 1 --unit s
```

4.5 Evaluating STL in MATLAB/Simulink

Now we describe how RTAMT monitors can be used in MATLAB/Simulink models and illustrate the API in Fig. 8. The integration is done by inserting the `online_monitor` block into an existing Simulink model. The monitor is a MATLAB Level 2 S-function block that integrates RTAMT as explained in Sec. 3.4. The block has one input and one output port. In order to pass multiple input signals to the monitor via the single input port, all the input signals must be combined into a single array signal using the `multiplexer` block. The output of the block is the robustness signal, which can be used to visually inspect the robustness of the property but also for any other appropriate purpose.

5 Implementations of STL Extended Language

In this section, we describe how practitioners implement a new specification with the RTAMT library. We first sketch how to extend STL monitors with the IA-STL syntax and semantics. The aim is to reuse as much as possible of the RTAMT STL monitoring implementation and only add implementation where STL and IA-STL differ. We first identify and localize main differences between STL and IA-STL:

- Syntax: IA-STL extends STL with the ability to declare signal variables as *input* and *output*.
- Semantics: the only difference in the semantics of IA-STL compared to STL is the treatment of numerical predicates.

The syntax extension requires adding the *input* and *output* keywords to the lexer and adding an appropriate parsing rule for allowing to decorate variables with these keywords. To add the semantic extension, the practitioner must extend the visitor used to traverse the STL formula parse tree during its evaluation and overwrite the method that evaluates the numerical predicates.

[6] We recall that bfSTL is defined over real-valued variables. As a consequence, we allow bfSTL properties to refer to Python variables and attributes of type `int`, `long` and `float`

[7] We allow only pSTL and bfSTL. The pastification is applied in the background automatically when practitioners use bfSTL.

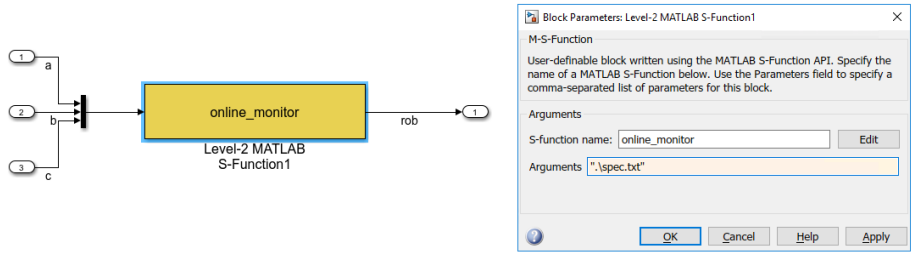


Fig. 8: Integration of RTAMT in MATLAB/Simulink.

The second scenario is the addition of a new operator to STL. In this case, practitioners also need to change both syntax and semantics. First, the name of the operator must be added as a reserved word to the lexer, as well as a new parsing rule that describes its signature. In contrast to the previous scenario, the visitor that parses the formula during evaluation must be extended with a new rule that provides the evaluation algorithm for the new operator. Since the RTAMT library provides support functionalities in abstract classes, practitioners can implement a new specification monitor quickly and easily. Python enables fast prototyping whereas C++ offers fast computation; the practitioner may choose according to their need.

6 Experiments

We make experiments for RTAMT from various perspectives and in different environments: RTAMT unit testing, ROS, and MATLAB/Simulink. All experiments are run on Intel® i9-10900K, 3.7 [GHz], 10 cores, and 128 [GB] RAM on Ubuntu 18.04.

6.1 Comparison of Computational Efficiency

In this section, we empirically evaluate the computational efficiency of the RTAMT in different settings.

We first show how RTAMT scales in the size of the input trace and the length of formulas in the offline monitors of the Python back-end. The results are depicted in Fig. 9 and show the difference of the average calculation time between discrete-time and dense-time monitors.

Fig. 10 compares the computation time of the online monitors of the C++ back-end against the Python back-end. To compare the two algorithms, we use for the experiment the STL specification $\mathbf{G}_{[0,k]}(a + b \geq -2)$ where k is the upper bound on the timing modality of the always operator, which we vary between 100 and 1 million. The outcomes clearly demonstrate more than approximately 10 times better efficiency of the C++ back-end, especially for large upper bounds in temporal modalities.

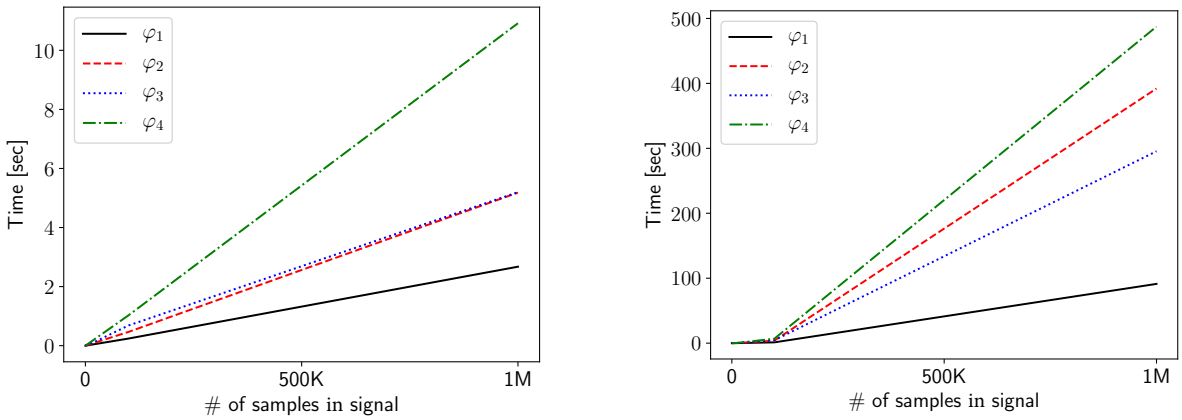
Those experiences guide the choice of the monitors for practitioners. However, even the slowest case which is dense-time of the Python back-end calculates approximately 0.5 [ms] per a sample and has a good enough real-time performance.

6.2 HSR Simulator in ROS

We now show that RTAMT not only monitors safety specification, but also enables fault localization on the component level in an autonomous mobile system in an ROS environment. This experiment follows authors' previous work [15] which is based on assume-guarantee fashion.

We applied RTAMT4ROS to Human Support Robot (HSR) [16] (Fig. 11), provided by Toyota as a robot platform. Physically, HSR has 8 Degrees of Freedom (DoF), combining the 3 DoF of the mobile base, 4 DoF of the arm and 1 DoF of the torso lift (Table 1), as well as several (Light Detection And Ranging) LiDARs, along with Stereo and monocular cameras. The platform supports ROS Gazebo [17] simulator. We applied RTAMT4ROS to the simulation.

Fig. 12b provides an overview of the target's software architecture. The simple perception, planner, and controller layers constitute the basic and abstract architecture in an autonomous



(a) Discrete-time

(b) Dense-time

Fig. 9: Scaling to the number of samples in the input signal and the length of formulas in the offline monitors: $\varphi_1 \equiv req \geq 3$, $\varphi_2 \equiv req \geq 3 \rightarrow gnt \geq 3$, $\varphi_3 \equiv req \geq 3 \rightarrow \mathbf{F}_{[0,5]} gnt \geq 3$, $\varphi_4 \equiv \mathbf{H}((req \geq 3) \rightarrow \neg(req \geq 3)\, \mathbf{U}_{[0,5]}(gnt \geq 3))$. Each experiment was repeated 50 times, and we report the average computation time.

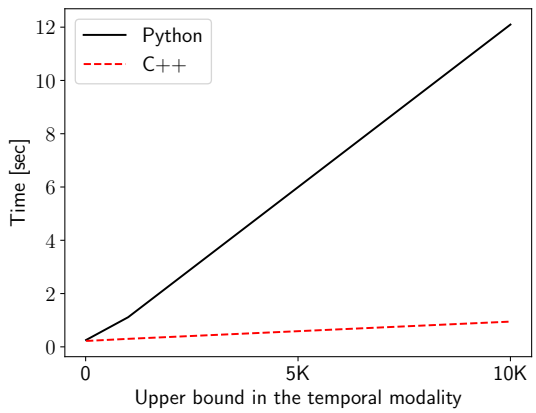


Fig. 10: Comparison between Python and C++ in the online monitors.

mobile system [18] based on the ROS software platform. Here is a brief description of each subsystem.

Perception In an actual robot, such as a range sensor, radar and vision-based perception are commonly used with sensor fusion techniques such as Kalman-filters or Particle-filters for detection and localization [19]. Specifically, the HSR perception layer has 2D grid map-based localization from the ROS navigation package [20]. In formalism, the perception recognizes statuses: ego position $\mathbf{x} = (x, y, \theta)$ where 2D position x, y, and angular position θ, agents' position $\mathcal{A} = \{\mathbf{x}_{\mathbf{a}_1}, \ldots, \mathbf{x}_{\mathbf{a}_{|\mathcal{A}|}}\}$, static obstacles' position $\mathcal{O} = \{O_1, \ldots, O_{|\mathcal{O}|}\}$, and prohibited regions $\mathcal{B} = \{B_1 \ldots, B_{|\mathcal{B}|}\}$ from ground truth of $\mathbf{x}^*$, $\mathcal{A}^*$, $\mathcal{O}^*$, and $\mathcal{B}^*$ from real-world plant with error ε.

Planner This layer decides its own path $\mathcal{W} = \{\mathcal{L}_1, \ldots, \mathcal{L}_{|\mathcal{W}|}\}$ based on the previous perception layer's recognition $\mathbf{x}$, $\mathcal{A}$, $\mathcal{O}$, and $\mathcal{B}$. Generally, the robot has a global path planner and local path planner. The former maps an overview of a path from a high-level graph similar to a static car-navigation map; the latter generates a motion path that treats more closed and dynamic obstacles. In the HSR planner layer, for simplicity, we omit the global path planner and deploy only Rapidly-exploring Random Trees (RRTs) [21] as a local path planner to reach a goal.

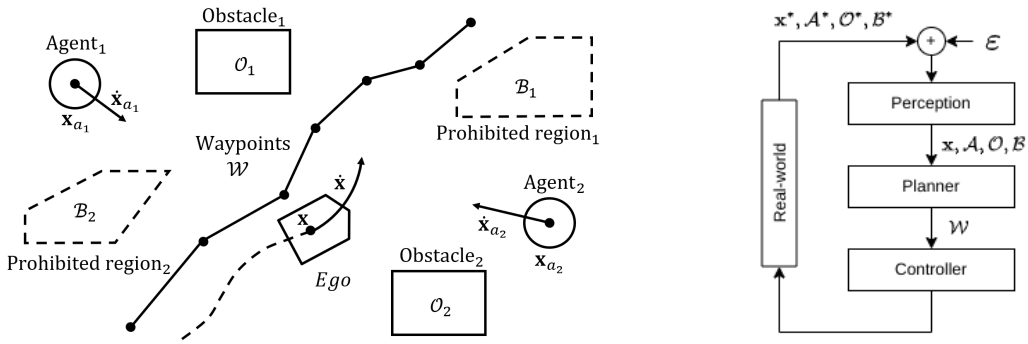
Controller Typically, this layer handles low-level hardware such as vehicle base control and actuator control based on $\mathcal{W}$ from the previous planner subsystem. Because it is a low-level control, generally a PID controller yields adequate results. In the case of HSR, a PID-based path follower is implemented which tracks way-points, minimizing error between current position $\mathbf{x}$ and $\mathcal{W}$ from the planner.

Based on this architecture, we define safety specifications in the system and subsystems. The most important system-level safety specification is

Fig. 11: HSR.

Table 1: HSR basic specification.

Size	Footprint	ϕ450 [mm]
	Height (min/max)	1,047/1,392 [mm]
	Freedom	8
	Weight	About 40 [kg]
Base	Drive system	Omnidirectional
	Environment	Indoor step 10 [mm]
		Climbing 7 [deg]
	Max speed	0.22 [m/s]
Hoisting	Stroke	690 [mm]
	Max speed	150 [mm/s]
Arm	Length	About 600 [mm]
	Movable height	0˜1,350 [mm]
	Movable depth	450 [mm]
	Payload	0.5 [kg]
Gripper	Grip speed	<0.4 [s]
	Max force	About 40 [N]
	Grip range	135 [mm]
	Max suction force	About 4 [N]



(a) Variable map. (b) Architecture.

Fig. 12: HSR system: (a) ego position $\mathbf{x}$, agents' position $\mathcal{A} = \{\mathbf{x}_{\mathbf{a}_1}, \ldots, \mathbf{x}_{\mathbf{a}_{|\mathcal{A}|}}\}$, static obstacles' position $\mathcal{O} = \{O_1, \ldots, O_{|\mathcal{O}|}\}$, and prohibited regions $\mathcal{B} = \{B_1 \ldots, B_{|\mathcal{B}|}\}$ (b) ground truth of $\mathbf{x}^*$, $\mathcal{A}^*$, $\mathcal{O}^*$, and $\mathcal{B}^*$ from real-world plant with error ε.

a collision prohibition.

$$\varphi_{sys} \equiv \bigwedge_{T_i \in \mathcal{A}, \mathcal{O}, \mathcal{B}} \mathbf{G}_{[0,\tau]} \left(\mathbf{Dist}(\mathbf{x}^*, T_i^*) > C_{coll} \right) \tag{1}$$

Where **Dist** is the Euclidean distance and τ is the time window of the formula. Simply $\mathbf{G}$ expresses "all time" the distance between the ego and others should be greater than some threshold. We use ground truths $\mathbf{x}^*$, $\mathcal{A}^*$, $\mathcal{O}^*$, and $\mathcal{B}^*$ while take an advantage of using a simulator since $\mathbf{x}$, $\mathcal{A}$, $\mathcal{O}$, and $\mathcal{B}$ have recognition errors of those.

The role of the perception subsystem is to recognize other obstacles and agents as much as possible. In other words, we may check the discrepancy between any recognition and the ground truth as perception specifications:

$$\varphi_{per} \equiv \bigwedge_{T_i \in \mathbf{x}, \mathcal{A}, \mathcal{O}, \mathcal{B}} \mathbf{G}_{[0,\tau_1]} \mathbf{F}_{[0,\tau_2]} \left(err(T_i, T_i^*) < \varepsilon_{per} \right) \tag{2}$$

Where *err* is a perception error metrics in each specific domain. We use $\mathbf{G}\,\mathbf{F}$ to give a tolerance of the violation with the time window τ_1. τ_2 is the duration time which permits the error since a small perception error is expected and it does not affect the system issue immediately.

The planner subsystem should generate a reasonable path $\mathcal{W}$ which does not cause collision with others:

$$\varphi_{plan} \equiv \bigwedge_{T_i \in \mathcal{A},\mathcal{O},\mathcal{B}} \mathbf{G}_{[0,\tau]} \left(\mathbf{Dist}(\mathcal{W}, T_i) > C_{plan}\right) \tag{3}$$

Finally, the controller subsystem should follow the reference as a low level controller:

$$\varphi_{con} \equiv \mathbf{G}_{[0,\tau_1]}\mathbf{F}_{[0,\tau_2]} \left(err(\mathcal{W}, \mathbf{x}) < \varepsilon_{con}\right) \tag{4}$$

Now we share the results of evaluation of system safety and subsystem level of fault localization when the system has a violation, based on the formal specifications above (Fig. 14). Each of the specification parameters are $\tau = 3$ [sec], $\varepsilon = 0.3$ [m] in Eq. 1 $\tau_1 = 7$, $\tau_2 = 5$ [sec], $\varepsilon = 0.1$ [m] in Eq. 2, $\tau = 3$ [sec], $\varepsilon = 0.2$ [m] in Eq. 3, and $\tau_1 = 7$, $\tau_2 = 5$ [sec], $\varepsilon = 0.1$ [m/s] in Eq. 4. Since all properties are bfSTL, in order to evaluate online, we applied pastification for all properties. First, the system all green case is shown in Fig. 14a, where we see all the specifications on both the system and subsystem levels satisfy the robustness of Eq. 1, 2, 3, 4 all the time. On the other hand, Fig. 14b shows a violation of system specification and subsystem level of fault localization. The planner is intentionally injected with a malfunction, the RRT being cut-off and given the wrong way points, causing a collision with a desk. The experiment shows that planner specification Eq. 3 is violated due to fault injection, so that the system specification in Eq. 1 is violated either. In reality, it is difficult for a designer to detect a system fail based on a subsystem fail because there are many subsystems, and a subsystem failure does not always cause a system failure. However, we assume a designer can at least detect a suspicious subsystem when the system failure. This STL assumption-based fault localization approach can also find a system issue caused by perception or controller failure [15].

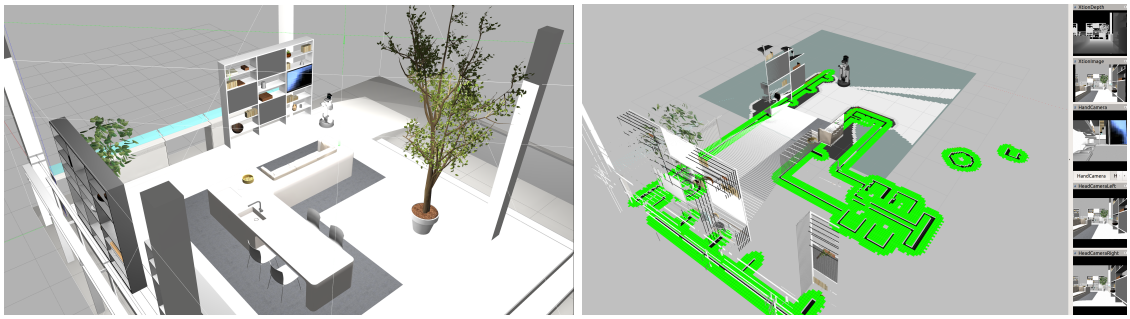
6.3 MATLAB/Simulink Model

In this section, we show how the RTAMT library integrated into MATLAB/Simulink and its resulting robustness monitors can be used to support Model-Based Development (MBD) of CPS. We explore two scenarios using the aircraft elevator control system (AECS), a model that illustrates MBD of a fault-tolerant control system (Fig. 15).These flight control systems are typically associated to the horizontal tails of the aircraft. They are responsible for the control of the aircraft's pitch. The system incorporates safety-relevant redundant components, namely: (1) four hydraulic actuators (two per elevator), three hydraulic controllers for driving the actuators, two Primary Flight Control Units (PFCUs) and two control modules for each actuator, one providing full range control law and limited range (degraded) control law. AECS has one input, the Pilot Command (PC), and two observable outputs, the position of left and right actuators (LEP and REP). The main requirement intuitively states that the actuators must follow PC within reasonable time and error. We translate the requirement for the LEP (the requirement for the REP is similar) to the following IA-STL specification:

$$\mathbf{G}\left(\uparrow (pc \geq m) \rightarrow \mathbf{G}_{[0,T]}\mathbf{F}_{[0,t]}(|pc - lep| \leq n)\right),$$

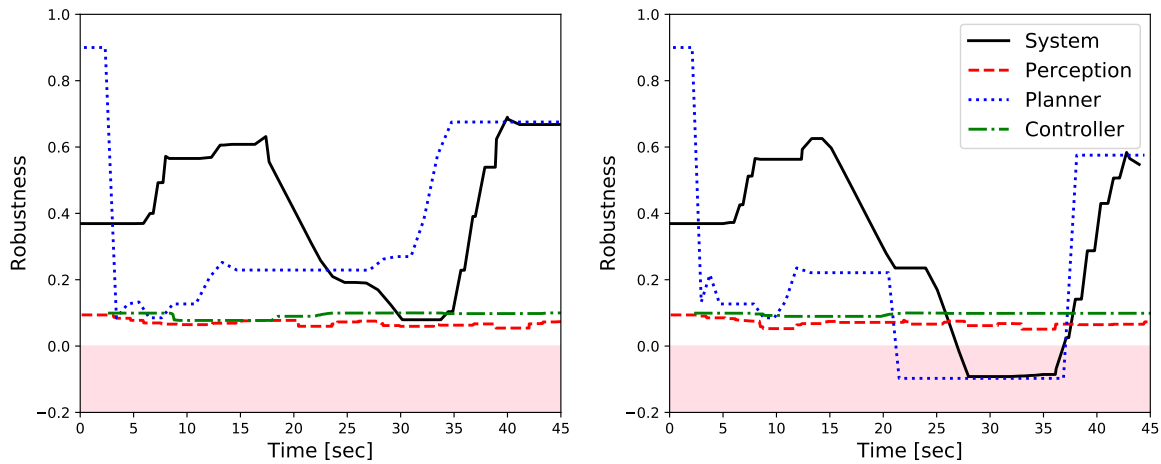
where m, n, t and T are constants.

We use RTAMT to do *sensitivity analysis* and *falsification testing*, two common approaches supported by tools like Breach and S-TaLiRo. Sensitivity of the model robustness to its STL requirements is studied by uniformly varying input parameters, simulating the model for each combination and monitoring the simulation outcomes against the requirements. In this case study, we test the system against PC step signals. A step signal is fully characterized by two parameters, the frequency and the amplitude of individual steps in the signal. We varied the frequency of the step input between 0.2 and 1 and the amplitude between 1 and 1.75, generating in total 16 simulations, one for each (amplitude, frequency) concrete pair. For every simulation, we evaluated how robust is the system behavior with respect to the bounded stabilization requirement for that specific input signal. Fig. 16a shows a heat-map visualizing the outcomes. This graphical representation of the results helps understanding how different input parameters affect the requirements and enable identifying critical parameter regions.



(a) Gazebo view as a ground truth. (b) Rviz view as a robot perception view.

Fig. 13: HSR simulator based on ROS Gazebo.



(a) The case of all systems satisfying safety specification. https://youtu.be/6xFHhv-F9A4 (b) The case of violation because of the planner fault. https://youtu.be/nGJ8siD8Fgk

Fig. 14: HSR experiment on fault localization with system (Eq. 1) and subsystem safety specifications (Eq. 2-4) in STL with RTAMT: (a) shows all specifications are satisfied (robustness is greater than zero). (b) shows the system specification is finally violated (robustness is below zero) and we can expect the planner subsystem is the cause because it fails before the system since other perception and planner subsystems satisfy the specification.

Given an STL specification and the system model, falsification testing aims to identify an input signal that results in the violation of the requirement. It does so by framing the test generation as a global optimization problem. The global optimizer (used as an off-the-shelf black-box component in this context) has the task of minimizing the system robustness to a requirement by finding appropriate inputs that steer the system in that direction, where the quantitative output of the monitor is used as the objective function. In essence, the global optimizer performs an iterative search in the space of input signals, where each iteration aims at steering the system towards lower robustness. The outcome is shown in Fig. 16b – every point (x, y) corresponds to a simulation of the system where x is the iteration number and y is the robustness of the system behavior to the requirement as measured by the monitor.

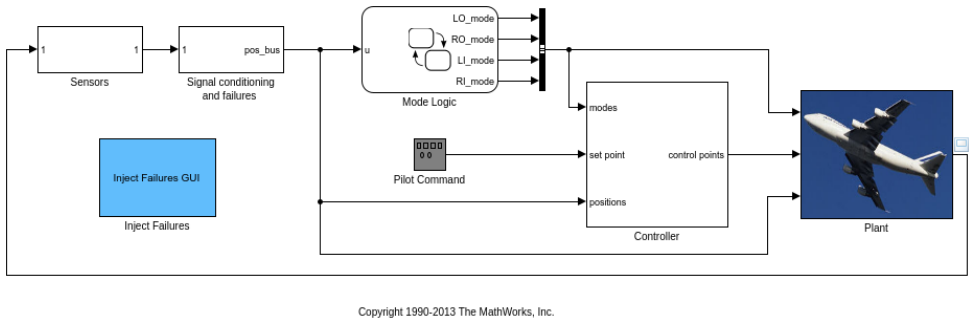
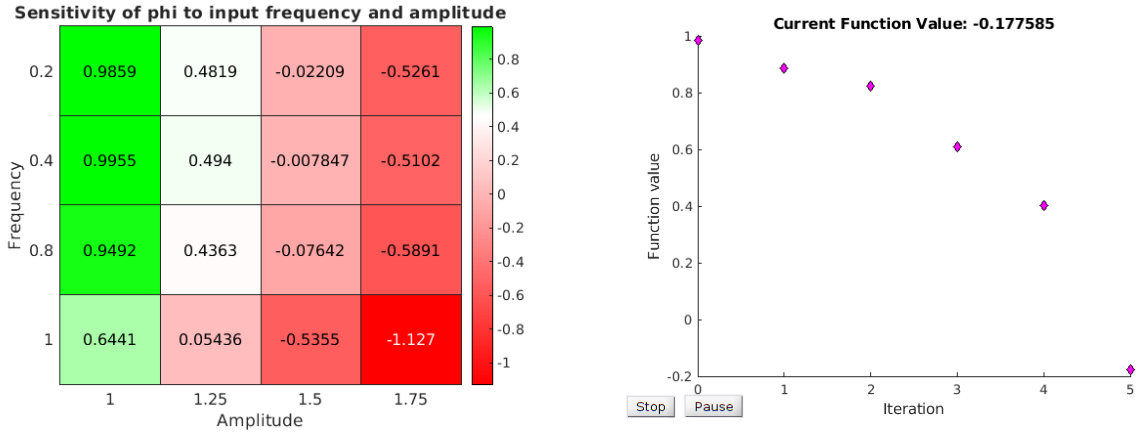


Fig. 15: Aircraft Elevator Control System (AECS).



(a) Sensitivity analysis.

(b) Falsification testing.

Fig. 16: AECS experiment: (a) a heat-map of the robustness on the input parameters. (b) a gradual reduction of the robustness to falsify the requirement by a global optimizer.

7 Related Work

Linear Temporal Logic (LTL) [22] is a well-known formal specification language for describing temporal properties of reactive systems. Signal Temporal Logic (STL) [6] is an extension of LTL that allows reasoning about real-valued signals and their real-time properties. Originally, both LTL and STL are interpreted using *qualitative* semantics – an observed behavior either satisfies or violates its specification. Fainekos and Pappas [23, 24] first proposed equipping temporal logic with *spatial* quantitative semantics, based on the *infinity norm*. Donzé and Maler's [25] adapt spatial robustness to STL and extend it with a complementary notion of *time robustness*. Intuitively, time robustness indicates how much an observed signal needs to be shifted in time in order to satisfy or violate the specification. Ferrère et al. developed an efficient algorithm for quantitative evaluation of STL [13], rendering the approach practical.

A combined notion of time and space robustness is proposed in [26]. Akazaki extended STL with averaged temporal operators in [27], which allow to quantify *how often* a temporal operator is satisfied within a bounded interval.

Evaluating temporal specifications with quantitative semantics was implemented in S-TaLiRo [28] and Breach [29] tools. These works established the theoretical background for our library, which implements STL with infinity-norm quantitative semantics.

Runtime verification of declarative specifications has been extensively studied. Havelund and

Rosu [30] proposed a dynamic programming algorithm to monitor past-time Linear Temporal Logic (ptLTL) specifications. Our inspiration to strive for a simple and elegant online monitoring procedure partly arose from this seminal paper. Reinbacher et al. propose in [31, 32] synthesizable hardware monitors from past-time Metric Temporal Logic (ptMTL) interpreted over discrete-time. Similarly, Jakšić et al. developed monitors that could be implemented in hardware for bounded-future Signal Temporal Logic (bfSTL). Reinbacher et al. [33] proposed monitoring procedures for Mission-time Linear Temporal Logic (MLTL), a variant of discrete-time MTL. The monitoring procedure consists of two types of observers: a synchronous one that yields a 3-valued instant abstraction of the satisfaction check, and an asynchronous one that makes this abstraction concrete at a later (bounded) time. This approach also allows a probabilistic estimation of the system health using a Bayesian network on top of the synchronous observers. This monitoring procedure was implemented in the R2U2 tool [34, 35]. More recently, MLTLM [36] was proposed as a language that allows to define temporal specifications over variables of multiple type. Our library supports quantitative semantics and makes a distinction between inputs and outputs.

Another relevant field of research is *stream runtime verification (SRV)*, in which runtime monitors are specified by describing operations that transform input streams of data to output streams of data. Precursor ideas of collecting statistics over execution traces can be found in [37]. Lola [38] proposes a SRV approach for synchronous systems. Lola 2.0 [39] extends the original language with parameterization and multiple temporal time scales. RTLola [40] and Tessla [41] provide support for specifying and monitoring real-time properties of reactive systems. Finally, Striver [42] provides a more general language that enables expressing other real-time monitoring languages. SRV approaches are powerful and typically more general than logic-based approaches. Temporal logic formalism, such as LTL and STL, can be often expressed with SRV. However, this comes at the price of requiring the user to explicitly encode the temporal logic semantics and thus losing a useful layer of abstraction between the user and the underlying formalism.

The problem of online robustness monitoring was studied in [43, 44]. Dokhanchi et al. [43] proposed an online monitoring approach which required a model of the system to make predictions about its behavior. In [44], Deshmukh et al. proposed an interval-based approach of online evaluation that allows estimating the minimum and the maximum robustness with respect to both the observed prefix and unobserved trace suffix. An alternative approach to online monitoring proposed a specification language that relaxes the causality restriction and allows the output to depend on a bounded amount of future input [45]. Algebraic approaches to runtime verification of temporal logic equipped with quantitative semantics were studied in [46] and [47]. These papers considered alternative ways to address online (qualitative and quantitative) monitoring of temporal logic specifications.

Pattern matching of Timed Regular Expressions (TREs) [48, 49] has been a lively area of research in recent years. Ulus et al. [50] proposed an offline pattern matching procedure for TRE with qualitative semantics and interpreted over continuous time, utilizing a novel matching approach that used operations on 2-dimensional zones. Ulus et al. [51] proposed an online variant of the procedure, also using the idea of derivatives adapted to the continuous-time. These pattern matching algorithms were implemented in the MONTRE tool [52]. Unlike previous automata-based matching algorithms for TREs which have been developed in [53–55], our work is centered on temporal logics rather than regular expressions.

8 Future Work and Conclusions

We presented RTAMT, a library for generating online and offline monitors from declarative specifications. We have integrated RTAMT into ROS as RTAMT4ROS and applied it to robotic applications as a robotics domain experiment. We have also integrated RTAMT into MATLAB/Simulink and applied it to AECS as a control domain experiment. The tool provides a flexible and modular library that enables practitioner specific specification language implementation.

In order to support process management that handles both time and event driven cases, we

assumed dense-time as a perfect continuous clock, a realistic assumption in many applications. As a consequence, we have extended RTAMT with an event-driven online monitoring algorithm for bfSTL in which sensor measurements will be allowed to arrive at any point on the dense-time axis.

The current library infrastructure already allows the monitor to continuously publish the computed robustness degree. This will enable decentralizing and distributing monitors. We will investigate both the theory and the implementation of distributed RTAMT monitors, especially in the context of robotic applications.

We plan to provide support for additional specification languages, but also for other semantic extensions of existing languages such as STL with weighted edit distance semantics.

Finally, we will further evaluate the library and the tool in other application domains. Actually, online monitoring of declarative specifications has various potential use-cases in the broad CPS domain and not only in robotics. Typical controller properties are naturally expressed in specification languages such as STL [56]. For Advanced Driver Assistance Systems (ADAS), there has been work on defining critical scenarios for pre-crash safety systems [57]. For self-driving systems, Responsibility-Sensitive Safety (RSS) [58] has been proposed, based on a clear definition by a math formula. Since RSS is not a temporal logic-based definition, it would be interesting to encode it in STL [59].

We plan to evaluate our library in such scenarios, both in a simulation environment and during actual in-field physical testing. Search-based testing [28, 29] also uses formalisms such as STL/MTL as a cost function to find input vectors that steer the system to property violation. This approach defines test generation as an optimization problem. In each iteration, the simulator executes the new input vector and the outcomes are evaluated in an offline fashion. We plan to explore whether RTAMT can make this process more efficient and dynamic by evaluating the outputs after each simulation step, adapting online search. VerifAI [60] is a simulation-based verification environment for AI-based applications that includes search-based testing which is implemented in Python and is one candidate for the application of RTAMT. Finally, we will consider potential applications of RTAMT in the verification and validation of perception systems based on neural networks. Several formal verification targets [59, 61, 62], approaches [63–66], and specifications [67] have been recently proposed. Unexpected failure of such perception systems can be checked during operation using our library.

References

[1] Lee, E.A., Seshia, S.A.: Introduction to embedded systems: A cyber-physical systems approach (2016)

[2] Mitra, S.: Verifying cyber-physical systems: A path to safe autonomy (2021)

[3] Alur, R.: Principles of cyber-physical systems (2015)

[4] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA Workshop on Open Source Software, vol. 3, p. 5 (2009). Kobe, Japan

[5] Sha, L.: Using simplicity to control complexity. IEEE Software (4), 20–28 (2001)

[6] Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, 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, Grenoble, France, September 22-24, 2004, Proceedings, pp. 152–166 (2004)

[7] Ferrère, T., Nickovic, D., Donzé, A., Ito, H., Kapinski, J.: Interface-aware signal temporal logic. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019, pp. 57–66 (2019)

[8] Nickovic, D., Yamaguchi, T.: RTAMT: online robustness monitors from STL. In: Automated Technology for Verification and Analysis - 18th International Symposium, ATVA

2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings, pp. 564–571 (2020). https://doi.org/10.1007/978-3-030-59152-6_34. https://doi.org/10.1007/978-3-030-59152-6_34

[9] Maler, O., Nickovic, D., Pnueli, A.: On synthesizing controllers from bounded-response properties. In: Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings, pp. 95–107 (2007)

[10] Jaksic, S., Bartocci, E., Grosu, R., Kloibhofer, R., Nguyen, T., Nickovic, D.: From signal temporal logic to FPGA monitors. In: 13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015, pp. 218–227 (2015)

[11] Parr, T.: The definitive antlr 4 reference (2013)

[12] Henzinger, T.A., Manna, Z., Pnueli, A.: What good are digital clocks? In: International Colloquium on Automata, Languages, and Programming, pp. 545–558 (1992). Springer

[13] Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Computer Aided Verification (CAV), pp. 264–279 (2013)

[14] Nickovic, D., Maler, O.: AMT: A property-based monitoring tool for analog systems. In: Formal Modeling and Analysis of Timed Systems, 5th International Conference, FORMATS 2007, Salzburg, Austria, October 3-5, 2007, Proceedings, pp. 304–319 (2007)

[15] Yamaguchi, T., Hoxha, B., Prokhorov, D., Deshmukh, J.V.: Specification-guided software fault localization for autonomous mobile systems. In: 2020 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE), pp. 1–12 (2020). IEEE

[16] Yamamoto, T., Terada, K., Ochiai, A., Saito, F., Asahara, Y., Murase, K.: Development of human support robot as the research platform of a domestic mobile manipulator. ROBOMECH Journal **6**(1), 4 (2019)

[17] Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, Sendai, Japan, pp. 2149–2154 (2004)

[18] Urmson, C., Anhalt, J., Bagnell, D., Baker, C., Bittner, R., Clark, M., Dolan, J., Duggins, D., Galatali, T., Geyer, C., *et al.*: Autonomous driving in urban environments: Boss and the urban challenge. Journal of Field Robotics **25**(8), 425–466 (2008)

[19] Thrun, S., Burgard, W., Fox, D.: Probabilistic robotics (2005)

[20] Marder-Eppstein, E., Berger, E., Foote, T., Gerkey, B., Konolige, K.: The office marathon: Robust navigation in an indoor office environment. In: International Conference on Robotics and Automation (2010)

[21] Kuffner, J.J., LaValle, S.M.: Rrt-connect: An efficient approach to single-query path planning. In: Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065), vol. 2, pp. 995–1001 (2000). IEEE

[22] Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), pp. 46–57 (1977). ieee

[23] Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications. In: Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers, pp. 178–192 (2006)

[24] Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theor. Comput. Sci. **410**(42), 4262–4291 (2009). https://doi.org/10.1016/j.tcs.2009.06.021

[25] Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Formal Modeling and Analysis of Timed Systems (FORMATS), pp. 92–106 (2010)

[26] Abbas, H., Mittelmann, H., Fainekos, G.: Formal property verification in a conformance testing framework. In: 2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 155–164 (2014). IEEE

[27] Akazaki, T., Tasuo, I.: Time robustness in MTL and expressivity in hybrid system falsification. In: Computer Aided Verification, 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2011, Proceedings (2015 (to appear))

[28] Annpureddy, Y., Liu, C., Fainekos, G.E., Sankaranarayanan, S.: S-taliro: A tool for temporal logic falsification for hybrid systems. In: Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings, pp. 254–257 (2011)

[29] Donzé, A.: Breach, A toolbox for verification and parameter synthesis of hybrid systems. In: Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings, pp. 167–170 (2010)

[30] Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 342–356 (2002). Springer

[31] Reinbacher, T., Függer, M., Brauer, J.: Real-time runtime verification on chip. In: Proc. of RV 2012. LNCS, vol. 7687, pp. 110–125 (2013). https://doi.org/10.1007/978-3-642-35632-2-13

[32] Reinbacher, T., Függer, M., Brauer, J.: Runtime verification of embedded real-time systems. Formal Methods in System Design **44**(3), 230–239 (2014)

[33] Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 357–372 (2014). Springer

[34] Schumann, J., Moosbrugger, P., Rozier, K.Y.: Runtime analysis with R2U2: A tool exhibition report. In: Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings, pp. 504–509 (2016)

[35] Rozier, K.Y., Schumann, J.: R2U2: tool overview. In: RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA, pp. 138–156 (2017)

[36] Hariharan, G., Kempa, B., Wongpiromsarn, T., Jones, P.H., Rozier, K.Y.: MLTL multi-type (MLTLM): A logic for reasoning about signals of different types. In: Software Verification and Formal Methods for ML-Enabled Autonomous Systems - 5th International Workshop, FoMLAS 2022, and 15th International Workshop, NSV 2022, Haifa, Israel, July 31 - August 1, and August 11, 2022, Proceedings, pp. 187–204 (2022)

[37] Finkbeiner, B., Sankaranarayanan, S., Sipma, H.: Collecting statistics over runtime executions. In: Runtime Verification 2002, RV 2002, FLoC Satellite Event, Copenhagen, Denmark, July 26, 2002, pp. 36–54 (2002). https://doi.org/10.1016/S1571-0661(04)80576-0. https://doi.org/10.1016/S1571-0661(04)80576-0

[38] D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington,

Vermont, USA, pp. 166–174 (2005)

[39] Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings, pp. 152–168 (2016). Springer

[40] Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time stream-based monitoring. arXiv preprint arXiv:1711.03829 (2017)

[41] Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: Tessla: temporal stream-based specification language. In: Formal Methods: Foundations and Applications: 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26–30, 2018, Proceedings 21, pp. 144–162 (2018). Springer

[42] Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: Runtime Verification: 18th International Conference, RV 2018, Limassol, Cyprus, November 10–13, 2018, Proceedings 18, pp. 282–298 (2018). Springer

[43] Dokhanchi, A., Hoxha, B., Fainekos, G.: Online monitoring for temporal logic robustness. In: International Conference on Runtime Verification, pp. 231–246 (2014). Springer

[44] Deshmukh, J.V., Donzé, A., Ghosh, S., Jin, X., Juniwal, G., Seshia, S.A.: Robust online monitoring of signal temporal logic. Formal Methods in System Design **51**(1), 5–30 (2017)

[45] Mamouras, K., Wang, Z.: Online signal monitoring with bounded lag. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **39**(11), 3868–3880 (2020). https://doi.org/10.1109/TCAD.2020.3013053

[46] Jaksic, S., Bartocci, E., Grosu, R., Nickovic, D.: An algebraic framework for runtime verification. IEEE Trans. on CAD of Integrated Circuits and Systems **37**(11), 2233–2243 (2018). https://doi.org/10.1109/TCAD.2018.2858460

[47] Mamouras, K., Chattopadhyay, A., Wang, Z.: Algebraic quantitative semantics for efficient online temporal monitoring. In: Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I, pp. 330–348 (2021)

[48] Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. Journal of ACM **49**(2), 172–206 (2002)

[49] Asarin, E., Caspi, P., Maler, O.: A Kleene theorem for timed automata. In: Logic in Computer Science (LICS), pp. 160–171 (1997)

[50] Ulus, D., Ferrère, T., Asarin, E., Maler, O.: Timed pattern matching. In: Formal Modeling and Analysis of Timed Systems (FORMATS), pp. 222–236 (2014)

[51] Ulus, D., Ferrère, T., Asarin, E., Maler, O.: Online timed pattern matching using derivatives. In: Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, pp. 736–751 (2016)

[52] Ulus, D.: Montre: A tool for monitoring timed regular expressions. In: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I, pp. 329–335 (2017)

[53] Waga, M., Hasuo, I.: Moore-machine filtering for timed and untimed pattern matching. IEEE Trans. on CAD of Integrated Circuits and Systems **37**(11), 2649–2660 (2018)

[54] Waga, M., Hasuo, I., Suenaga, K.: Efficient

online timed pattern matching by automata-based skipping. In: Formal Modeling and Analysis of Timed Systems - 15th International Conference, FORMATS 2017, Berlin, Germany, September 5-7, 2017, Proceedings, pp. 224–243 (2017)

[55] Waga, M., Hasuo, I., Suenaga, K.: MONAA: A tool for timed pattern matching with automata-based acceleration. In: 3rd Workshop on Monitoring and Testing of Cyber-Physical Systems, MT@CPSWeek 2018, Porto, Portugal, April 10, 2018, pp. 14–15 (2018)

[56] Kapinski, J., Jin, X., Deshmukh, J., Donze, A., Yamaguchi, T., Ito, H., Kaga, T., Kobuna, S., Seshia, S.: St-lib: A library for specifying and classifying model behaviors. Technical report, SAE Technical Paper (2016)

[57] Najm, W.G., Smith, J.D., Yanagisawa, M., et al.: Pre-crash scenario typology for crash avoidance research. Technical report, United States. National Highway Traffic Safety Administration (2007)

[58] Koopman, P., Osyk, B., Weast, J.: Autonomous vehicles meet the physical world: Rss, variability, uncertainty, and proving safety. In: International Conference on Computer Safety, Reliability, and Security, pp. 245–253 (2019). Springer

[59] Hekmatnejad, M., Yaghoubi, S., Dokhanchi, A., Amor, H.B., Shrivastava, A., Karam, L., Fainekos, G.: Encoding and monitoring responsibility sensitive safety rules for automated vehicles in signal temporal logic. In: Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design, pp. 1–11 (2019)

[60] Dreossi, T., Fremont, D.J., Ghosh, S., Kim, E., Ravanbakhsh, H., Vazquez-Chanlatte, M., Seshia, S.A.: Verifai: A toolkit for the design and analysis of artificial intelligence-based systems. arXiv preprint arXiv:1902.04245 (2019)

[61] Rong, G., Shin, B.H., Tabatabaee, H., Lu, Q., Lemke, S., Možeiko, M., Boise, E., Uhm, G., Gerow, M., Mehta, S., *et al.*: Lgsvl simulator: A high fidelity simulator for autonomous driving. In: 2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC), pp. 1–6 (2020). IEEE

[62] Vitelli, M., Chang, Y., Ye, Y., Wołczyk, M., Osiński, B., Niendorf, M., Grimmett, H., Huang, Q., Jain, A., Ondruska, P.: Safetynet: Safe planning for real-world self-driving vehicles using machine-learned policies. arXiv preprint arXiv:2109.13602 (2021)

[63] Liu, C., Arnon, T., Lazarus, C., Barrett, C., Kochenderfer, M.J.: Algorithms for verifying deep neural networks. arXiv preprint arXiv:1903.06758 (2019)

[64] Tuncali, C.E., Fainekos, G., Ito, H., Kapinski, J.: Simulation-based adversarial test generation for autonomous vehicles with machine learning components. In: 2018 IEEE Intelligent Vehicles Symposium (IV), pp. 1555–1562 (2018). IEEE

[65] Date, Y., Baba, T., Hoxha, B., Yamaguchi, T., Prokhorov, D.: Application of simulation-based methods on autonomous vehicle control with deep neural network: Work-in-progress. In: 2020 International Conference on Embedded Software (EMSOFT), pp. 1–3 (2020). IEEE

[66] Ghosh, S., Pant, Y.V., Ravanbakhsh, H., Seshia, S.A.: Counterexample-guided synthesis of perception models and control. In: 2021 American Control Conference (ACC), pp. 3447–3454 (2021). IEEE

[67] Dreossi, T., Ghosh, S., Sangiovanni-Vincentelli, A., Seshia, S.A.: A formalization of robustness for deep neural networks. arXiv preprint arXiv:1903.10033 (2019)