Fast Robust Monitoring for Signal Temporal Logic with Value Freezing Operators (STL*)

Bassem Ghorbel
Colorado State University
USA
bassem@colostate.edu

Vinayak S. Prabhu
Colorado State University
USA
vinayak.prabhu@colostate.edu

Abstract—Researchers have previously proposed augmenting Signal Temporal Logic (STL) with the value freezing operator in order to express engineering properties that cannot be expressed in STL. This augmented logic is known as STL*. The previous algorithms for STL* monitoring were intractable, and did not scale formulae with nested freeze variables. We present offline discrete-time monitoring algorithms with an acceleration heuristic, both for Boolean monitoring as well as for quantitative robustness monitoring. The acceleration heuristic operates over time intervals where subformulae hold true, rather than over the original trace sample-points. We present experimental validation of our algorithms, the results show that our algorithms can monitor over long traces for formulae with two or three nested freeze variables. Our work is the first work with monitoring algorithm implementations for STL* formulae with nested freeze variables.

I. INTRODUCTION

In the context of Cyber-Physical Systems (CPS) and Control, *Signal Temporal Logic* (STL) has found wide adoption as a trace property specification formalism [1], [2], [3], [4], [5], [6], [7], [8], [9]. STL, which can be seen as a flavor of Metric Temporal Logic (MTL) [10], allows specification of properties such as "if the temperature rises above 100 degree Celsius at any point in time, then it will within 5 time units, fall below 50 degree Celsius, and stay below 50 degree Celsius for at least 2 time units". Such specifications incorporate both a temporal aspect (e.g., "within 5 time units") as well as a signal constraints aspect (e.g., "temperature > 100"). Two key problems over temporal logics for CPS are (1) *Boolean monitoring*: checking whether a given trace satisfies a temporal logic specification; and (2) *Robustness monitoring*: defining a quantitative measure of how well a given trace satisfies a temporal logic specification, and computing this numerical value. Tools such as STaLiRo, Breach, FALSTAR, and FalCAuN use such robustness monitoring procedures for STL for test generation in order to falsify STL specifications [1], [11], [8], [12].

While STL demonstrates the utility of temporal logics for verification, control, and testing of CPS, it is unable to express commonly occurring properties in biological and engineering systems, such as oscillatory properties, as has been noted by researchers [13], [14], [15]. A natural manner to increase the expressive power of STL is to add *freeze quantification*, which allows the capture of a signal value into a freeze variable, to be used later in the trace for comparison [13]. Freeze quantification was first introduced in the context of temporal operators as *time* freeze quantification in [16], and the resulting increase in expressive power was

proved in [17]. The work [13] introduced the logic STL* and showed how *value* freeze quantification in the *signal domain* enabled specification of properties which are believed to be outside of the scope of STL. We illustrate the value freeze operator. Consider the requirement: "*At some future time (during interval I_1), there is a local maximum over 5s, then at another future time (during interval I_2), there is a local minimum over 5s*". This requirement can be written in STL* as $\varphi_0 = \Diamond_{I_1} s_{*1}. \big(\big(\Box_{[0,5]} s \leq s_{*1} \big) \wedge \Diamond_{I_2} s_{*2}. \big(\Box_{[0,5]} s \geq s_{*2} \big) \big)$. The freeze variable s_{*1} freezes the signal value at some point in interval I_1, and this frozen value is accessed as s_{*1} for the local maxima check in $\Box_{[0,5]} s \leq s_{*1}$, and similarly for freeze variable s_{*2}.

However, the increased expressivity of freeze quantifiers incurs a price on monitoring algorithms — which is to be expected since the monitoring problem for temporal freeze quantifier augmented MTL is PSPACE hard [16], [17]. An alternative mechanism to increase temporal logic expressivity is by adding *first order* quantification [18]; however first order quantification similarly makes monitoring intractable in the general case [19]. Orthogonal to freeze quantification, monitoring algorithms also depend on whether pointwise semantics of traces (a trace being a sequence of signal values) or whether continuous time semantics (with traces being completed using linear or piecewise constant interpolation from sampled values) are used; the resulting impact on algorithm intricacy can be seen even in STL [20], [11]. The Boolean monitoring algorithm in [13] for STL* uses continuous time semantics with linear interpolation; it is complex, and involves manipulation of polygons. Even in the case of a single value freeze operator, and even for *approximate* monitoring in an attempt to make the problem tractable, the algorithm in [13] remains complex. Due to the complicated nature of the algorithm which involves manipulating polygons, [13] did not obtain a precise complexity bound: it was only shown that "steps of the algorithm has at most polynomial complexity to the number of polygons and the number of polygons grows at most polynomially in each of the steps". Their monitoring experiments showed scalability limitations – over an hour of running time for signals containing 100 time-points, and even restricted to STL* formulae containing only one active freeze variable.

The work of [21] proposes a STL* robustness monitoring algorithm for pointwise semantics, but the algorithm calls – for every possible binding of a freeze operator to a frozen signal value – a subroutine which has a $O(|\pi|^2)$ dependence where $|\pi|$ is the trace length. If $|V|$ is the number of freeze quantifiers

used, there are $|\pi|^{|V|}$ freeze operator bindings, resulting in an overall algorithm time complexity of $O(|\pi|^{|V|+2})$, and hence the procedure is not tractable even for one freeze variable. The experiments in [21] are over specialized formulae: only one freeze variable, no until operator, and no nested temporal operators within the scope of the freeze variable; this allows the use of a special algorithm, which is not described. The recent work of [22] examined the Boolean monitoring problem for the fragment of STL* in which any subformula could contain at most one "active" freeze variable (i.e., no nested freeze variables). For this one variable fragment of STL*, they presented an efficient Boolean monitoring algorithm in the pointwise semantics scaling to trace lengths of 100k. That work did not address the robustness monitoring problem. In the present work, we build upon [22], lift the one variable limitation, and examine the Boolean as well as the robustness monitoring problem for the full logic of STL* with nested value freeze variables.

Our Contributions. Our main contributions are:

(*I*) We investigate which engineering properties require more than one active value freeze variable, and we present natural requirements using two and three freeze variables which we conjecture cannot be specified in one variable STL*.

(*II*) We present offline Boolean monitoring algorithms for full STL* in the pointwise semantics, for both uniformly, and non-uniformly time-sampled traces. Our algorithms use an *acceleration* technique based on the work from [22]. We show that the acceleration technique can be used for Boolean monitoring for full STL*. Suppose we have $|V|$ freeze variables in a formula φ. In order to check whether a trace π satisfies the formula, we need to iterate over the trace, and at each trace point, there are $|\pi|^{|V|}$ possibilities for the bindings of the $|V|$ freeze variables as each freeze variable can be bound to the signal value at any sample point. This gives us a search space of approximately $|\pi|^{|V|+1}$ for monitoring. We show that we need *not* iterate over all timepoints in the trace for each subformula and for every freeze variable binding: it turns out that in most cases we can iterate over *intervals* rather than individual timepoints – these intervals are the partitions of the time-stamps where the relevant subformulae have a true value throughout. This results in a monitoring complexity of $O\big(|\pi|^{|V|} \cdot \max\big(\log(|\pi|), |\varphi| \cdot |\operatorname{intvl}(\varphi)|\big)\big)$ for a uniformly sampled trace in practice, where $|\operatorname{intvl}(\varphi)|$ is the resulting number of such intervals for any sub-formula of φ (and similarly for non-uniformly sampled traces). The acceleration technique leverages the idea that these intervals do not change much from one freeze binding to the next for realistic traces where signal values do not vary wildly from one timepoint to the next. In practice, $|\operatorname{intvl}(\varphi)| << |\pi|$. Thus, our acceleration heuristic reduces the exponent in $|\pi|^{|V|+1}$ by one (at the expense of $\log(|\pi|)$).

(*III*) We present offline robustness monitoring algorithms for full STL*. Unfortunately, the intervals idea of Boolean monitoring cannot directly be used for value computation, as from freeze binding to one sampled signal value to the next, the quantitative robustness *value* would indeed change. However, we show that the acceleration heuristic can be used for the robustness *decision problem* which asks whether the robustness value is less than or equal to some given threshold. We achieve the robustness value computation by binary

search over a conservative robustness range. We show how to compute this conservative range given a formula and a trace. We also give a non-interval monitoring algorithm, improving upon the algorithm from [21] by a careful handing of the until operator; this improves the $|\pi|^{|V|+2}$ time complexity factor from [21] to $|\pi|^{|V|+1}$.

(*IV*) We obtain time complexity bounds for our algorithms.

(*V*) We implement our algorithms and present experimental results. We show that with the accelerated algorithms, monitoring for two nested freeze variables remains tractable – Boolean monitoring for traces of size 10k takes about 3 minutes, and about 17 minutes for robustness value computation (with the final robustness value estimate being within 2% of the actual value). We believe two nested freeze variables suffice to capture most engineering properties of interest in STL*. We are also able to monitor for three nested freeze variables over traces of size 500 in about 2.5 minutes, and for traces of size 1k in about 22 minutes for Boolean monitoring. We note that [13], [21] did not implement monitoring for two freeze variables in their experiments as their algorithms do not scale to two freeze variables. Thus, ours is the *first* work which presents implemented monitoring algorithms for STL* formulae with nested freeze variables, both for Boolean monitoring as well as for robustness value computation. The technical report [23] contains details omitted due to lack of space from this paper.

Related Work. Logics augmented with frequency constructs have recently been proposed for property specification in the frequency domain [24], [25]. Freeze quantification enables expanded expression of properties in the time domain [26]. In general, it is well known that both time and frequency domain analyses provide useful information about signals. Efficient algorithms for *time* freeze quantification are presented in [27], [28], [29].

II. VALUE-FREEZING SIGNAL TEMPORAL LOGIC

Signals/Traces. Let $\dim \in \mathbb{N}_{>0}$, a $\mathbb{R}^{\dim}$ valued *signal* or a *trace* is a pair (σ, τ), where $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_{|\pi|-1}$ is a finite sequence of elements from $\mathbb{R}^{\dim}$, and $\tau = \tau_0, \tau_1, \ldots, \tau_{|\pi|-1}$ are the corresponding timestamps from $\mathbb{R}_+$. The signal value at timestamp τ_i is $\sigma_i \in \mathbb{R}^{\dim}$ and i is a position index. The k-th signal dimension of $\sigma = \langle a^1, \ldots, a^{\dim} \rangle$, namely a^k, is denoted $\sigma^k = \sigma_0^k, \sigma_1^k, \ldots, \sigma_{|\pi|-1}^k$. We require the times to be monotonically increasing, that is $\tau_i < \tau_{i+1}$ for all i. If $\tau_i = i \cdot \Delta$ for some $\Delta > 0$, the traces are said to be *uniformly sampled*; *non-uniformly sampled* otherwise.

To reduce notation clutter, we use $\overline{x} = x^1, x^2, \ldots, x^{\dim}$ for any variable x.

Definition 1 (STL* Syntax). Given a signal arity $\dim$, and a finite set of *freeze variables* $\{s_*^1, \ldots, s_*^{\dim}\}$ for each signal dimension $1 \leq k \leq \dim$, the syntax of value-freezing signal temporal logic (STL*) is defined as follows:

- $f(\overline{s}) \sim 0$; and $\quad f_1(\overline{s}) \sim f_2(\overline{s_*})$.
- $\neg\varphi$; and $\varphi_1 \wedge \varphi_2$; and $\varphi_1 \vee \varphi_2$.
- $\square_I \varphi$; and $\lozenge_I \varphi$; and $\varphi_1 \mathcal{U}_I \varphi_2$.
- $s_*^k.\varphi$.

where $s^k \in \{s^1, \ldots, s^{\dim}\}$ is a *signal variable* (s^k refers to the k-th signal dimension), $f(\overline{s}) \sim 0$ are signal predicates,

$f_1(\overline{s}) \sim f_2(\overline{s_*})$ are signal constraints, f, f_1 and f_2 are arbitrary functions, and $I = [a, b]$ is an interval where a and b are positive reals, s_*^k is a *freeze variable* corresponding to the signal-value freeze operator "s_*^k.", and $\sim \in \{<, >, \leq, \geq\}$ is the standard comparison operator.

The freeze operator "s_*^k." binds the current k-th signal dimension value to the frozen value s_*^k in the signal constraints. $\square$

The original work in [13] allowed only affine functions for f, f_1 and f_2; but as our treatment is in discrete-time, we can handle arbitrary computable functions.

Note: We can freeze the same signal dimension multiple times in an STL* formula. In that case, we use a subscript $h \in \mathbb{N}_{>0}$ to indicate which frozen value refers to which signal-value freeze operator. Thus, $s_{*h_1}^k$ and $s_{*h_2}^k$ for $h_1 \neq h_2$ are considered different freeze variables. When we freeze a signal dimension only once in a STL* formula, we omit the superscript $h = 1$ (check example 2).

Definition 2 (Semantics). Let $\pi = (\sigma_0, \tau_0), (\sigma_1, \tau_1), \ldots, (\sigma_{|\pi|-1}, \tau_{|\pi|-1})$ be a finite timed signal of arity dim. For a given environment $\mathcal{E} : V \to \mathbb{R}$ binding freeze variables to signal values, and position index $0 \leq i \leq |\pi| - 1$, the satisfaction relation $(\pi, i, \mathcal{E}) \models \varphi$ for an STL* formula φ of arity dim (with freeze variables in V) is defined as follows.

- $(\pi, i, \mathcal{E}) \models f(\overline{s}) \sim 0$ iff $f(\overline{\sigma_i}) \sim 0$.
- $(\pi, i, \mathcal{E}) \models \neg\varphi$ iff $(\pi, i, \mathcal{E}) \not\models \varphi$.
- $(\pi, i, \mathcal{E}) \models \varphi_1 \overset{\vee}{\wedge} \varphi_2$ iff $(\pi, i, \mathcal{E}) \models \varphi_1 \overset{\text{or}}{\text{and}} (\pi, i, \mathcal{E}) \models \varphi_2$.
- $(\pi, i, \mathcal{E}) \models \Diamond_{[a,b]}\varphi$ iff $\exists j, \tau_j \in \tau_i + [a, b]$, s.t. $(\pi, j, \mathcal{E}) \models \varphi$.
- $(\pi, i, \mathcal{E}) \models \Box_{[a,b]}\varphi$ iff $\forall j, \tau_j \in \tau_i + [a, b]$, we have $(\pi, j, \mathcal{E}) \models \varphi$.
- $(\pi, i, \mathcal{E}) \models \varphi_1 \mathcal{U}_{[a,b]}\varphi_2$ iff $(\pi, j, \mathcal{E}) \models \varphi_2$ for some $j \geq i$ with $\tau_j \in \tau_i + [a, b]$ and $(\pi, k, \mathcal{E}) \models \varphi_1 \; \forall i \leq k < j$.
- $(\pi, i, \mathcal{E}) \models s_*^k.\varphi$ iff $(\pi, i, \mathcal{E}[s_*^k := \sigma_i^k]) \models \varphi$; where $\mathcal{E}[s_*^k := \sigma_i^k]$ denotes the environment $\mathcal{E}'$ defined as $\mathcal{E}'(x) = \mathcal{E}(x)$ for $x \neq s_*^k$, and $\mathcal{E}'(s_*^k) = \sigma_i^k$.
- $(\pi, i, \mathcal{E}) \models f_1(\overline{s}) \sim f_2(\overline{s_*})$ iff $f_1(\overline{\sigma_i}) \sim f_2(\mathcal{E}(s_*^1), .., \mathcal{E}(s_*^{\dim}))$.

We say trace π satisfies STL* formula φ if $(\pi, 0, \mathcal{E}[\equiv \sigma_0]) \models \varphi$ where $\mathcal{E}[\equiv \sigma_0]$ denotes the freeze environment where all freeze variables s_*^k are mapped to their corresponding σ_0^k values. $\square$

Example 1. We consider a single dimension signal s and we freeze it twice ($h = 1$ and $h = 2$) in the following formula: $\varphi_0 = \Diamond_{I_1} s_{*1}.(\Box_{[0,5]}s \leq s_{*1} \wedge \Diamond_{I_2} s_{*2}.(\Box_{[0,5]}s \geq s_{*2}))$. The requirement of φ_0 is: *"at some time in the future (during I_1), there is a local maximum over 5s, then at another time in the future (during I_2), there is a local minimum over 5s "*. $\square$

To reduce notation clutter, we will simply write $(i, \mathcal{E})$ instead of $(\pi, i, \mathcal{E})$ in the remainder of this paper since for any given STL* formula, we will be using the same trace π. We use the phrase i^{th} *instantiation* of a freeze variable s_*^k to mean the environment $\mathcal{E}$ where the freeze variable s_*^k is assigned the value σ_i^k.

III. EXPRESSIVENESS OF STL*

Authors in [13] introduced STL* with multiple freeze variables but did not give any requirement with more than one freeze variable. In this section, we present some interesting specifications that require more than just a single freeze variable. We provide four engineering properties that require STL* expressiveness with nested freeze variables.

Example 2 (Running Example). Eventually e_1 happens and after that, eventually, e_2 happens and 2-time units after that, the values of s are always within 20 % of the average of the value of s when e_1 happened and the value of s when e_2 happened (Stabilization of s around not known multiple values in advance). e_1 and e_2 can be any signal predicates: $\varphi_1 =$

$$\Diamond\left(e_1 \wedge s_{*1}.\left(\Diamond\left(e_2 \wedge s_{*2}.\Box_{[2,T]}s \in [0.8\frac{s_{*1} + s_{*2}}{2}, 1.2\frac{s_{*1} + s_{*2}}{2}]\right)\right)\right). \quad \square$$

Example 3. Eventually the value of s^1 is greater than 5 (at an unknown moment t_1) and after that, eventually the value of s^1 is greater than 10 (at an unknown moment t_2), and after that, eventually the value of s^2 is greater than the value of s^2 at t_1 plus the value of s^2 at t_2 until s^1 is less than 5: $\varphi_2 =$

$$\Diamond\left(s^1 > 5 \wedge s_{*1}^2.\Diamond\left(s^1 > 10 \wedge s_{*2}^2.\Diamond\left((s^2 > s_{*1}^2 + s_{*2}^2)\mathcal{U}s^1 < 5)\right)\right)\right). \quad \square$$

Previous STL* work used only linear constraints (and predicates) of the form $\sum a_i s^i + \sum b_i s_*^i \sim r$ where a_i, b_i and r are constants, while we do not impose such restrictions and we are free to use any type of functions.

Example 4. Check if s is a rectangular pulse signal with unknown pulse value: $\varphi_3 =$

$$\Box s_{*1}.\left(|s_{*1} - s| \leq \epsilon \, \mathcal{U}\left(|s_{*1} - s| \geq \Delta \wedge s_{*2}.(|s_{*2} - s| \leq \epsilon \, \mathcal{U}|s_{*1} - s| \leq \epsilon)\right)\right)$$

where ϵ is an error threshold and Δ is the minimum pulse amplitude. We note that the ϵ value can be in function of the frozen values, for example, $\epsilon = 0.1 \times s_{*1}$.

To understand the logic behind the above formula φ_3, we look at Figure 1 (a): At any given time point (which is represented by the $\Box$ at the beginning), the first freeze variable s_{*1}, freezes the value 1 (or -1) and the future values of s must remain within that frozen value s_{*1} with an ϵ error *until* a sudden increase or decrease of the amplitude of s by at least Δ. When that happens, the second freeze variable s_{2*} freezes the current s value which is -1 (or 1) and the future values of s must remain within that frozen value s_{*2} with an ϵ error *until* s is equal to the first frozen value s_{*1} with an ϵ error.

Some would suggest that this requirement can be expressed using just STL*$_1$ as follows

$$\psi = \Box s.\left(|s_* - s| \leq \epsilon \, \mathcal{U}\left(|s_* - s| \geq \Delta \wedge (|s'| \leq \epsilon \, \mathcal{U} \, |s_* - s| \leq \epsilon)\right)\right)$$

where s' is the derivative of s. However, this formula would have problems for signals with noise. Using the derivative makes our formula look at the signal locally, and will not give an idea on how the signal behaves globally (Figure 1).

In addition, for a largerϵ threshold, ψ can end up accepting signals that should not be accepted, see Figure 2. $\square$

Example 5. Check if a signal s follows a repeating two stairs signal with unknown pulse values: $\varphi_4 =$

$$\Box s_{*1}.\left(|s_{*1} - s| \leq \epsilon \, \mathcal{U}\left(|s_{*1} - s| \geq \Delta \wedge s_{*2}.\left(|s_{*2} - s| \leq \epsilon \, \mathcal{U}(|s_{*2} - s| \geq \Delta\right)\right)\right.$$

$$\wedge \, s_{*3}.(|s_{*3} - s| \leq \epsilon \, \mathcal{U} \, |s_{*1} - s| \leq \epsilon)))\Big)\Big). \qquad \square$$

(a) Signal that satisfies φ_3 and ψ (b) Signal that satisfies φ_3, not ψ
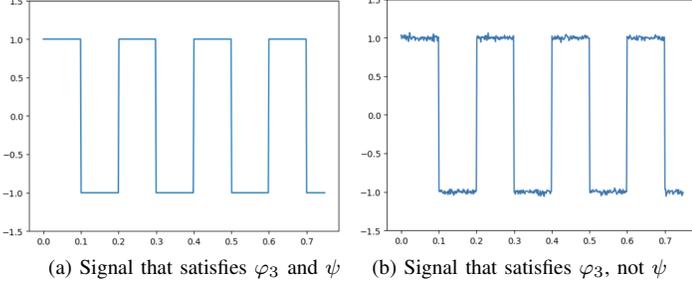
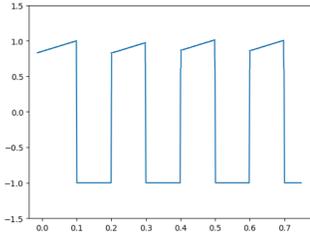Fig. 1: Signals that satisfy the rectangular pulse requirement

Fig. 2: Signal that can be accepted by ψ with large ϵ

IV. STL* SYNTAX TREES

Each STL* formula has a corresponding syntax tree that depicts the hierarchical syntactic structure of the formula. Our monitoring procedure will depend on this syntax tree.

The basic structure over which our algorithm will operate will be subtrees corresponding to various freeze variables. The following example explains subtrees, parents and roots.

Example 6. Consider formula φ_1 from Example 2. Figure 3 has its associated syntax tree. The formula subscripts correspond to a reverse topological sort of the syntax tree. The freeze variable ordering given by a reverse topological sort is $s_{*2} <_{\text{revtop}} s_{*1}$. The nodes in $\text{SubTree}_{\varphi_0}(s_{*2})$ are $\{\varphi_9, \varphi_{10}\}$. The nodes in $\text{SubTree}_{\varphi_0}(s_{*1})$ are $\{\varphi_5, \varphi_6, \varphi_7, \varphi_8\}$; and in $\text{TopSubTree}(\varphi)$ are $\{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$. $\text{SubTree}_{\varphi_0}(s_{*2}).\,\text{root} = \varphi_9$, $\text{SubTree}_{\varphi_0}(s_{*2}).\,\text{parent} = \varphi_8$, $\text{SubTree}_{\varphi_0}(s_{*1}).\,\text{root} = \varphi_5$, $\text{SubTree}_{\varphi_0}(s_{*1}).\,\text{parent} = \varphi_4$. $\square$

V. STL* BOOLEAN MONITORING ALGORITHM

The monitoring problem we consider in the paper is as follows: Given a trace π, and an STL* formula φ, do we have $(\pi, 0, \mathcal{E}[\equiv \sigma_0]) \models \varphi$? Before presenting our algorithm, we need first some notations.

Notation: For any STL* subformula ψ, we note:

- point(ψ): a list of length $|\pi|$ of TRUE and FALSE values where for each $0 \leq i \leq |\pi| - 1$, each value represents whether $(i, \mathcal{E}) \models \psi$ or not for a given environment $\mathcal{E}$.
- intvl(ψ): a list of intervals $[\tau_{a_1}, \tau_{a_2}], [\tau_{b_1}, \tau_{b_2}],$ $\ldots, [\tau_{z_1}, \tau_{z_2}]$, where $\tau_{a_1}, \tau_{a_2} \ldots \tau_{z_2}$ are timestamps and $a_1 \leq a_2 < b_1 \leq b_2 \cdots < z_1 \leq z_2$, each interval represents a sequence of TRUE's appearing in point(ψ).
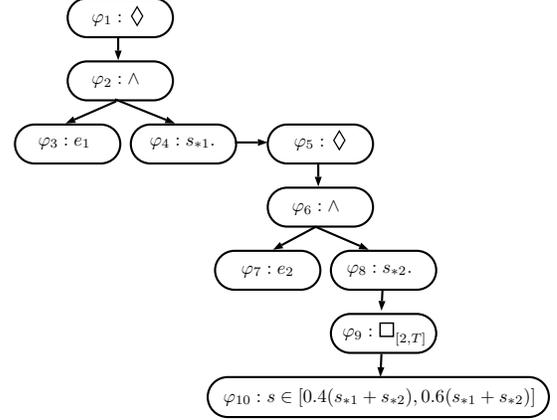

Fig. 3: Syntax Tree for Example 2

Example 7. Suppose we have the following trace $(\sigma_i, \tau_i), i \in [0, 10]$: (5,0), (3,1), (7,2), (-2,3), (-5,4), (3,5), (-1,6), (3,7), (4,8), (5,9), (6,10) and the subformula $\psi = s \geq 0$, then:

- point(ψ) = $[T, T, T, F, F, T, F, T, T, T, T]$.
- intvl(ψ) = $[0, 2], [5, 5], [7, 10]$. $\square$

A. Algorithm Overview

Suppose we have $|V|$ freeze variables $x_{*1}, \ldots, x_{*|V|}$ that are ordered in φ as follows: $x_{*1} > x_{*2} > \ldots > x_{*|V|}$, in other words, φ is of the form: $\ldots x_{*1}.(\ldots x_{*2}.(\ldots x_{*|V|}.(\ldots)))$ (we use x for the naming of the freeze variables instead of s_*^k to avoid any confusion with the subscript h and the signal dimensions superscript k).

We will need to consider all possible combinations of environments for all the freeze variables: $\mathcal{E}[x_{*1} := \sigma_{i_1}^{j_1}, x_{*2} := \sigma_{i_2}^{j_2}, \ldots, x_{*|V|} := \sigma_{i_{|V|}}^{j_{|V|}}]$ where $i_1 \ldots i_V$ are indices referring to timestamps each ranging from 0 to $|\pi| - 1$ and $j_k, 1 \leq k \leq |V|$ is the index of the signal dimension corresponding to the freeze variable x_{*k}. To better understand the idea behind this algorithm, let us consider φ_1 from example 2, where $x_{*1} = s_{*1}$ and $x_{*2} = s_{*2}$. We consider our first *environment group* which starts with the environment that freezes s_{*1} to σ_0 and s_{*2} to σ_0, for this first environment, the algorithm calculates the satisfaction relations for every $\psi \in \text{SubTree}_{\varphi_0}(s_{*2})$ for all the position indices i. Then, the algorithm considers the next environment which freezes s_{*1} to σ_0 and s_{*2} to σ_1 and calculates the same satisfaction relations for all the position indices $i \geq 1$. Similarly, the next environment would freeze s_{*1} to σ_0 and s_{*2} to σ_2 and so on $\ldots$ Once we reach the environment that freezes s_{*1} to σ_0 and s_{*2} to $\sigma_{|\pi|-1}$, the algorithm will calculate the satisfaction relation for every $\psi \in \text{SubTree}_{\varphi_0}(s_{*1})$ for all the position indices i. Then, our next *environment group* would start with the environment that freezes s_{*1} to σ_1 and s_{*2} to σ_1 and we repeat all the same steps above. The algorithm will go over all the possible environment groups.

For any $x_{*m} > x_{*n}$, if we want to calculate a satisfaction relation for any subformula for a given environment that binds x_{*m} to $\sigma_i^{j_m}$, we only need to consider bindings of x_{*n} to $\sigma_l^{j_n}$ where $i \leq l \leq |\pi| - 1$. That is why, in our previous example of φ_0, when we moved to the second environment group that freezes s_{*1} to σ_1, we started by freezing s_{*2} to σ_1 and not σ_0.

Here are the main steps of our algorithm:

- To be able to get the value of the satisfaction relation $(0, \mathcal{E}[\equiv \sigma_0]) \models \varphi$, we will need all the values of $(i, \mathcal{E}[\equiv \sigma_i]) \models \psi$ for every $\psi \in \mathrm{AST}(\varphi)$ and every $i \in [0, |\pi|-1]$.
- And for a given subformula $\varphi_1 \in \mathrm{SubTree}_\varphi(x_{*1})$, to get the values of $(i, \mathcal{E}[\equiv \sigma_i]) \models \varphi_1$, we will need all the values of $(j, \mathcal{E}[\equiv \sigma_j]) \models \psi$ for every $\psi \in \mathrm{AST}(\varphi_1)$ and every $j \in [0, |\pi| - 1]$.
- And for a given subformula $\varphi_2 \in \mathrm{SubTree}_\varphi(x_{*2})$, to get the value of $(j, \mathcal{E}[\equiv \sigma_j]) \models \varphi_2$, we will need all the values of $(k, \mathcal{E}[\equiv \sigma_k]) \models \psi$ for every $\psi \in \mathrm{AST}(\varphi_2)$ and every $k \in [0, |\pi| - 1]$.
- And so on ...

In general, for a formula with $|V|$ freeze variables, we will go over $|\pi|^{|V|}$ environments for each subformula $\psi \in \mathrm{SubTree}_\varphi(x_{*|V|})$, and for each $\psi' \in \mathrm{SubTree}_\varphi(x_{*|V|-1})$, we will be computing $|\pi|^{|V|-1}$ environments ...

The second major contribution of this work is based on the following idea: when trying to calculate $(i, \mathcal{E}) \models \psi$ for any given ψ and any environment $\mathcal{E}$, a naive algorithm would iterate over all the timestamps to calculate the different satisfaction relations. However, our algorithm iterates over the intervals in $\mathrm{intvl}(\psi)$ where in practice the size of $\mathrm{intvl}(\psi)$ (the size of $\mathrm{intvl}(\psi)$ is the number of intervals in $\mathrm{intvl}(\psi)$) is way smaller than the number of timestamps. This will give us the same results in a reduced number of computations. Let us consider example 7, and suppose we want to calculate $(i, \mathcal{E}) \models \square_{[1,2]}\varphi_0, \forall i$, instead of calculating 10 satisfaction relations (one for each i, our algorithm will calculate just 3 (one for each interval in $\mathrm{intvl}(\varphi_0)$, it has just 3 intervals). Also, in some cases, when a subformula ψ is either a signal constraint or of the form $s_*^k.\psi'$, we need to calculate $\mathrm{point}(\psi)$ (the vector $\mathrm{point}(\psi)$ represents $(i, \mathcal{E}) \models \psi, i \geq i'$ for a given $i' \in [0, |\pi|-1]$ and a given $\mathcal{E}$) and not just $\mathrm{intvl}(\psi)$. The nature of the trace π (pointwise semantics and discrete timestamps and not a continuous signal) is the main reason why we have to go over $\mathrm{point}(\psi)$ as a first step and not directly calculate $\mathrm{intvl}(\psi)$, in other words, we cannot calculate $\mathrm{intvl}(\psi)$ without calculating $\mathrm{point}(\psi)$ first, for ψ of these forms. For the case of a signal constraint, we try to update a limited number of values in $\mathrm{point}(\psi)$ and not iterate over all values of i.

Data Structures

- Timestamps array: Array of size $|\pi|$ which have the timestamps values.
- Signal dimension array: For each signal dimension, we use a $|\pi|$ sized array to store the signal dimension values.
- $\mathrm{sorted}(\varphi_j)$: Doubly linked list, look section V-C for more details.
- $\mathrm{point}(\varphi_j)$: Array of size $|\pi|$.
- $\mathrm{intvl}(\varphi_j) = [\mathrm{start}_j(0), \mathrm{end}_j(0)], \ldots, [\mathrm{start}_j(n), \mathrm{end}_j(n)]$.
- start_j and end_j: Arrays of size $n \leq |\pi|$ each.
- flip_j: Array of size $|\pi|$ for each signal constraint φ_j.

B. `Compress` *Algorithm*

Given input $\mathrm{point}(\varphi_j)$ and integer i, `Compress` computes $\mathrm{intvl}(\varphi_j)$ using the values in $\mathrm{point}(\varphi_j)$ from position i.

C. *Calculating* $\mathrm{sorted}(\varphi')$

Given any $\varphi' = f_1(\bar{s}) \sim f_2(x_{*1}, x_{*2}, \ldots, x_{*|V|})$ signal constraint, our algorithm needs to calculate $\mathrm{sorted}(\varphi')$. The

values in $\mathrm{sorted}(\varphi')$ only depend on the function f_1 and the values of $\sigma_i^k, i \in [0, |\pi| - 1]$ where k refers to the kth signal dimension for every signal dimension called by the function f_1. The algorithm calculates $f_1(\bar{s})$ for the different i values, sorts it and stores it in $\mathrm{sorted}(\varphi')$. Let us consider an example where $f_1 = s^1 + s^2 - 3$. The values of s^1 for the different timestamps are $2, 5, 7, 1, 9$ and the values of s^2 are $8, 2, -3, 4, 1$. Figure 4 shows the corresponding doubly linked list $\mathrm{sorted}(\varphi')$ and the links between $\mathrm{sorted}(\varphi')$ and the trace values (keep track of original position indices before sorting).
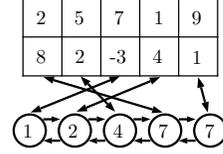

Fig. 4: s^1, s^2 dimension arrays and $\mathrm{sorted}(\varphi')$

D. *Main Algorithm*

Algorithm 1: STL* Monitor Algorithm

Input: $\mathrm{AST}(\varphi)$, π
Output: $\mathrm{intvl}(\varphi_1)$
1   $\mathrm{intvl}(\varphi') \leftarrow \texttt{Compress}(\mathrm{point}(\varphi'), 0), \forall \varphi'$ signal pred.
2   **for** *each sig constraint* $\varphi' = f_1(\bar{s}) \sim f_2(x_{*1}, \ldots, x_{*|V|})$ **do**
3     $\mathrm{sorted}(\varphi') \leftarrow$ sorted $f_1(\bar{s})$ values
4   Rec-STL* $(1, 0)$
5   **if** $|\mathrm{TopSubTree}(\varphi)| \neq 1$ **then**
6   **for** *each subformula* $\varphi_j \in \mathrm{TopSubTree}(\varphi)$ **do**
7       $\mathrm{intvl}(\varphi_j) \leftarrow \texttt{ComputeIntervals}(\varphi_j, 0)$
8   **return** $\mathrm{intvl}(\varphi_1)$

Algorithm 2: Rec-STL*(k, t)

Input: k, t
1   **for** $i \leftarrow t$ *to* $|\pi| - 1$ **do**
2   **for** *each signal constraint* φ_j *in* $\mathrm{SubTree}(x_{*k})$ **do**
3   **if** $i = 0$ **then**
4   Calculate $\mathrm{flip}_j[0]$, $\mathrm{point}(\varphi_j)$;
5         $\mathrm{intvl}(\varphi_j) \leftarrow \texttt{Compress}(\mathrm{point}(\varphi_j), 0)$
6   **else**
        $\mathrm{intvl}(\varphi_j) \leftarrow \texttt{UpdateSignalConstraint}(\varphi_j, i)$;
7   **if** $k < |V|$ **then** Rec-STL* $(k + 1, i)$;
8   **for** $j \leftarrow \mathrm{SubTree}_\varphi(x_{*k}).\max$ *down to* $\mathrm{SubTree}_\varphi(x_{*k}).\min$ **do**
9   **if** φ_j *is not a sig constraint nor a sig predicate* **then**
10        $\mathrm{intvl}(\varphi_j) \leftarrow \texttt{ComputeIntervals}(\varphi_j, i)$
11   **if** $\mathrm{intvl}(\mathrm{SubTree}_\varphi(x_{*k}).\mathrm{root})$ *starts with* τ_i **then**
12       $\mathrm{point}(\mathrm{SubTree}_\varphi(x_{*k}).\mathrm{parent})[i] \leftarrow \mathrm{TRUE}$
13   **else** $\mathrm{point}(\mathrm{SubTree}_\varphi(x_{*k}).\mathrm{parent})[i] \leftarrow \mathrm{FALSE}$;
14   $\mathrm{intvl}(\mathrm{SubTree}_\varphi(x_{*k}).\mathrm{parent}) \leftarrow$ $\texttt{Compress}(\mathrm{point}(\mathrm{SubTree}_\varphi(x_{*k}).\mathrm{parent}), t)$

The first line of the STL* Monitor Algorithm calculates $\mathrm{intvl}(\varphi')$ for any φ' signal predicate, the second and third lines calculate $\mathrm{sorted}(\varphi')$ for any φ' signal constraint. Line

4 will call the `Rec-STL`[*] algorithm using arguments $(1,0)$ to calculate the values of $\text{intvl}(\varphi')$ for every subformula φ' in $\text{SubTree}_\varphi(x_{*k})$ for every freeze variable x_{*k} in φ (more details in the next section). The remaining lines in the algorithm (5 to 7) calculate the values of $\text{intvl}(\varphi')$ for every subformula φ' in $\text{TopSubTree}(\varphi)$, this is the case when all the subformulae of type $x_{*k}.\psi$ have been computed. Finally, the algorithm returns $\text{intvl}(\varphi_1)$.

Note that if $|\text{TopSubTree}(\varphi)| = 1$, then the top subtree has only one node of type $x_{*k}.\psi$, and $\text{intvl}(x_{*k}.\psi)$ has already been computed by `Rec-STL`[*] so there is nothing to do.

E. `Rec-STL`[*](k,t)

The `Rec-STL`[*](k,t) function calculates the values of $\text{intvl}(\varphi')$ for all subformulae φ' in $\text{SubTree}_\varphi(x_{*j})$, $k \leq j \leq |V|$ for the different instantiations of x_{*k} to $\sigma_i^{j_k}$ for $i \in [t, |\pi| - 1]$. More simply, the ultimate goal of each call of `Rec-STL`[*](k,t) is to calculate all the $|\pi| - 1 - t$ values of $\text{point}(\text{SubTree}_\varphi(x_{*k}).\text{parent})$ (Lines 11-13) and eventually transform it to $\text{intvl}(\text{SubTree}_\varphi(x_{*k}).\text{parent})$ (Line 14). Each value in $\text{point}(\text{SubTree}_\varphi(x_{*k}).\text{parent})$ represents $(i, \mathcal{E}[x_{*k} := \sigma_i^{j_k}]) \models \text{SubTree}_\varphi(x_{*k}).\text{parent}, t \leq i \leq |\pi| - 1$. the main idea behind the `Rec-STL`[*] algorithm is the following: in order to calculate $\text{intvl}(\varphi')$ for $\varphi' \in \text{SubTree}_\varphi(x_{*k})$ at the ith instantiation, we calculate $\text{intvl}(\varphi')$ for $\varphi' \in \text{SubTree}_\varphi(x_{*k+1})$ at the instantiations i', $i' \in [i, |\pi| - 1]$, that is why, the first time we call `Rec-STL`[*], we use arguments $(\mathbf{1}, \mathbf{0})$ which aim to compute the satisfaction relations of subformulae in $\text{SubTree}_\varphi(x_{*\mathbf{1}})$ (the first input $\mathbf{1}$ refers to the freeze variable $x_{*\mathbf{1}}$) at time points $i \in [\mathbf{0}, |\pi - 1|]$ (the second input $\mathbf{0}$ refers to the smallest i value the algorithm considers).

The pre-condition for a function call `Rec-STL`[*](k,t) is that $\text{intvl}(\varphi')$ for any φ' signal constraints, belonging to subtrees $\text{SubTree}_\varphi(x_{*l})$ for $l < k$, have already been computed for x_{*l} instantiated to $\sigma_t^{j_l}$. Before going through a technical explanation, let us take an example and suppose our STL[*] formula φ is of the form $\varphi = ...x_{*1}.(...x_{*2}.(...x_{*3}.(...)))$ where the dots can be any operators and φ has 3 freeze variables. The first call `Rec-STL`[*]$(1,0)$ will calculate $\text{intvl}(\varphi')$ for any φ' signal constraint corresponding to x_{*1} for the instantiation 0. Then, `Rec-STL`[*]$(1,0)$ will call `Rec-STL`[*]$(2,0)$ to calculate $\text{intvl}(\varphi')$ for any φ' signal constraint corresponding to x_{*2} for the instantiation 0, and afterwards, `Rec-STL`[*]$(3,0)$ is called to calculate $\text{intvl}(\varphi')$ for any signal constraints φ' corresponding to x_{*3} for the instantiation 0. Since x_{*3} is the last freeze variable in φ, `Rec-STL`[*]$(3,0)$ will continue to calculate $\text{intvl}(\varphi')$ for any subformula $\varphi' \in \text{SubTree}_\varphi(x_{*3})$ for all the instantiations of x_{*3} to $\sigma_i^{j_3}$ for $i \in [0, |\pi| - 1]$. Once that is done, we go back to `Rec-STL`[*]$(2,0)$ to calculate $\text{intvl}(\varphi')$ for any $\varphi' \in \text{SubTree}_\varphi(x_{*2})$ corresponding to x_{*2} instantiated to $\sigma_0^{j_2}$. Then, for $i = 1$ in `Rec-STL`[*]$(2,0)$, we calculate intvl for the signal constraints corresponding to x_{*2} for the instantiation 1 and call `Rec-STL`[*]$(3,1)$ which will calculate $\text{intvl}(\varphi')$ for any subformula $\varphi' \in \text{SubTree}_\varphi(x_{*3})$ again but this time for all instantiations of x_{*3} starting from 1 and so on ...

How `Rec-STL`[*] is called : `Rec-STL`[*]$(1,0)$ $\rightarrow$ `Rec-STL`[*]$(2,0) \rightarrow$ `Rec-STL`[*]$(3,0) \rightarrow$ `Rec-STL`[*]$(3,1) \ldots \rightarrow$ `Rec-STL`[*]$(3,|\pi|-1) \rightarrow$ `Rec-STL`[*]$(2,1) \rightarrow$ `Rec-STL`[*]$(3,1) \rightarrow$ `Rec-STL`[*]$(3,2) \ldots \quad \rightarrow \quad$ `Rec-STL`[*]$(3,|\pi| - 1) \ldots \quad \rightarrow$ `Rec-STL`[*]$(2,|\pi|-1) \rightarrow$ `Rec-STL`[*]$(3,|\pi|-1)$.

In general, for the instantiation of x_{*k} to $\sigma_i^{j_k}$ (Line 1), we calculate or update the values of $\text{point}(\varphi_j)$ and $\text{intvl}(\varphi_j)$ for every $\varphi_j \in \text{SubTree}_\varphi(x_{*k})$ signal constraint (Lines 2-6). Then, in a recursive way, the algorithm does the same thing for all the remaining freeze variables $x_{*k'}, k \leq k' \leq |V|$ instantiated to $\sigma_i^{j_{k'}}$ (Line 7). Once the algorithm reaches the final freeze variable $x_{*|V|}$, we already have $\text{point}(\varphi_j)$ and $\text{intvl}(\varphi_j)$ calculated for all φ_j signal constraints of the different freeze variables corresponding to all freeze variable instantiated to the corresponding $\sigma_i^{j_{k'}}$, and the algorithm calculates the values of $\text{intvl}(\varphi')$ for every subformula $\varphi' \in \text{SubTree}_\varphi(x_{*|V|})$ by calling `ComputeIntervals` (Lines 8-10) and assigns a value to $\text{point}(\text{SubTree}_\varphi(x_{*|V|}).\text{parent})[i]$ (Lines 11-13) $(\text{point}(\text{SubTree}_\varphi(x_{*|V|}).\text{parent})[i]$ represents the ith value in the vector $\text{point}(\text{SubTree}_\varphi(x_{*|V|}).\text{parent})$ corresponding to the parent node of $\text{SubTree}_\varphi(x_{*|V|}))$. Then, it instantiates $x_{*|V|}$ to the next value $\sigma_{i+1}^{j_{|V|}}$ (next iteration of the *for* loop in Line 1), updates the signal constraints (Line 6), and calculates $\text{intvl}(\varphi')$ for every subformula $\varphi' \in \text{SubTree}_\varphi(x_{*|V|})$ (Line 10) and adds a new value to $\text{point}(\text{SubTree}_\varphi(x_{*|V|}).parent)$ and so on until we finish with all the instantiations of $x_{*|V|}$ (Line 1). Once that is done, we go back to the previous call of `Rec-STL`[*]$(|V|-1,i)$ and calculate the values of $\text{intvl}(\varphi')$ for every subformula $\varphi' \in \text{SubTree}_\varphi(x_{*|V|-1})$ (and $\text{point}(\varphi')$ in case of a signal constraint). Then, $x_{*|V|-1}$ is instantiated to $i + 1$ and we call `Rec-STL`[*]$(|V|, i + 1)$ and so on.

F. `ComputeIntervals` *Algorithm*

In this section, we show how we compute, for a given environment $\mathcal{E}$, $\text{intvl}(\varphi')$ of subformula φ' with boolean or temporal operators. The idea is based on [30], we slightly modify it to make it work for pointwise semantics. Suppose we have two traces with different sampling rates. The first one, π_1, is uniformly sampled of length 100 and the sampling rate is 1 second. And the second one, π_2, is non-uniformly sampled and it has the following timestamps: $0, 1, 2, 4, 5, 7, 8, 10, 11, 13, 15, 17, 20, 25, 27, 30, 35$ and 40. And let us consider two signal predicates $\varphi_1 = s^1 \geq 5$ and $\varphi_2 = s^2 \leq 0$ such that $\text{intvl}(\varphi_1) = [2,10], [20,35]$ and $\text{intvl}(\varphi_2) = [7,15]$, for both traces π_1 and π_2.

Boolean operators: For Boolean operators, the computation is straightforward. We have the following:

○ For the uniformly sampled trace π_1:
- $\text{intvl}(\neg\varphi_1) = [0,1], [11,19], [36,99]$.
- $\text{intvl}(\varphi_1 \vee \varphi_2) = [2,15], [20,35]$.
- $\text{intvl}(\varphi_1 \wedge \varphi_2) = [7,10]$.

○ For the non-uniformly sampled trace π_2:
- $\text{intvl}(\neg\varphi_1) = [0,1], [11,17], [40,40]$.
- $\text{intvl}(\varphi_1 \vee \varphi_2) = [2,15], [20,35]$.
- $\text{intvl}(\varphi_1 \wedge \varphi_2) = [7,10]$.

For Boolean operators, computing $\text{intvl}(\varphi')$ takes $O(|\text{intvl}(\varphi')|)$.

Temporal operators: To treat temporal operators, we need to define the following $[a,b]$-*back shifting* operation:

Definition 3. Let $I = [m, n]$ and $[a, b]$ be intervals and k an index position. The $[a, b]$-back shifting of I, is
$$I \ominus [a, b] = [m - b, n - a]$$
We also define the trim of I, $\text{trim}^k(I)$, to be the largest possible interval $[\tau_i, \tau_j], k \le i, j \le |\pi - 1|$ included in I. $\quad\square$

Note 1: When we omit the superscript k, it means $k = 0$.

Note 2: For the trim operator, given a $\text{intvl}(\varphi)$ with $|\text{intvl}(\varphi)|$ intervals, if the trace is uniformly sampled (in other words, for a given timestamp, we know the next and previous timestamps in $O(1)$ time), we can calculate $\text{trim}(\text{intvl}(\varphi))$ in $O(|\text{intvl}(\varphi)|)$ time. However, if the trace is not uniformly sampled, calculating $\text{trim}(\text{intvl}(\varphi))$ takes $O(|\text{intvl}(\varphi)|.\log(|\pi|))$ where the $O(\log(|\pi|))$ is paid to find the largest possible interval $[\tau_i, \tau_j], k \le i, j \le |\pi| - 1$ included in I for each interval I in $\text{intvl}(\varphi)$ using binary search. Or, we can simply iterate over all the timestamps in π to find $\text{trim}(\text{intvl}(\varphi))$ since the intervals in $\text{intvl}(\varphi)$ are ordered. This makes calculating $\text{trim}(\text{intvl}(\varphi))$ takes $O(|\pi|)$. We use the exponential search algorithm, in case of non uniformly sampled trace, to reduce the complexity of calculating $\text{trim}(\text{intvl}(\varphi))$ to the best possible case which is $O\Big(\min\big(|\pi|, |\text{intvl}(\varphi)| \cdot \log(|\pi|)\big)\Big)$.

Eventually operator $\Diamond_{[a,b]}$: To calculate $\Diamond_{[a,b]}\varphi'$, we just do $\text{trim}(\text{intvl}(\varphi') \ominus [a, b])$. For example,
- For π_1, $\text{intvl}(\Diamond_{[1,3]}\varphi_1) = [0, 9], [17, 34]$.
- For π_2, $\text{intvl}(\Diamond_{[1,3]}\varphi_1) = [0, 8], [17, 30]$.

Until operator $\mathcal{U}_{[a,b]}$: For $\varphi\mathcal{U}_{[a,b]}\psi$, we will use the same claim used in [30] for STL formulae and can be generalized for STL* formulae.

Claim 1. Let $\varphi = \varphi_1 \vee \varphi_2 \cdots \vee \varphi_p$ and $\psi = \psi_1 \vee \psi_2 \cdots \vee \psi_q$ be two STL* subformula, each written as a union of unitary subformula (with a single interval). Then
$$\varphi\mathcal{U}_{[a,b]}\psi = \bigvee_{i=1}^{p}\bigvee_{j=1}^{q} \varphi_i\mathcal{U}_{[a,b]}\psi_j \quad\square$$

For each interval I in φ and J in ψ, we do the following: $\big((I \cap J) \ominus [a, b]\big) \cap I$. Then, we apply the trim operation to all intervals.

For example, let us consider first the uniformly sampled trace π_1: for $\varphi_1\mathcal{U}_{[2,4]}\varphi_2$,
(a) $[2, 10] \cap [7, 15] = [7, 10]$, $[7, 10] \ominus [2, 4] = [3, 8]$, $[3, 8] \cap [2, 10] = [3, 8]$ and
(b) $[20, 35] \cap [7, 15] = \emptyset$
$\Rightarrow \text{intvl}(\varphi_1\mathcal{U}_{[2,4]}\varphi_2) = [3, 8]$.
And, for the non-uniformly sampled trace π_2: we have $\text{intvl}(\varphi_1\mathcal{U}_{[2,4]}\varphi_2) = [4, 8]$.
- Uniformly sampled trace: This operation will take $O(|\text{intvl}(\varphi_1)| + |\text{intvl}(\varphi_2)|)$.
- Non uniformly sampled trace: This operation will take $O(\min(|\pi|, |\text{intvl}(\varphi_1)| + |\text{intvl}(\varphi_2)|.\log|\pi|))$.

G. UpdateSignalConstraint *Algorithm*

Let φ_j be a signal constraint, the main goal of this algorithm is to update the values of $\text{point}(\varphi_j)$ and $\text{intvl}(\varphi_j)$ for the instantiation $i+1$ given the values of $\text{point}(\varphi_j)$ and $\text{intvl}(\varphi_j)$ for the instantiation i.

To do so, the algorithm uses flip_j's to track which values should be updated in $\text{point}(\varphi_j)$ and $\text{intvl}(\varphi_j)$. $\text{flip}_j[i]$ is the position index where φ_j $\big(\text{in sorted}(\varphi_j)\big)$ switches values from TRUE to FALSE or the opposite in the ith instantiation. Here, if we interpret the signal constraint φ_j as a function of the frozen values, and since the values are sorted in $\text{sorted}(\varphi_j)$, we can see that $\text{flip}_j[i]$ represents a threshold for when we reach a value in $\text{sorted}(\varphi_j)$ for which φ_j is TRUE (resp. FALSE) for all the next values in $\text{sorted}(\varphi_j)$ and FALSE (resp. TRUE) for all the previous values. Given $\text{flip}_j[i]$ and $\text{flip}_j[i+1]$, it updates certain values in $\text{point}(\varphi_j)$ and $\text{intvl}(\varphi_j)$ (values corresponding to position indices between $\text{flip}_j[i]$ and $\text{flip}_j[i + 1] - 1$ in $\text{sorted}(\varphi)$). Further details can be found in [23].

VI. RUNNING EXAMPLE

In this section, we will go over the running steps of our algorithm. We will use a formula φ similar to φ_1 from our running example, we slightly modify the signal constraint (node φ_{10} in Figure 3) for the sake of simplicity:
$$\varphi = \Diamond\left(e_1 \wedge s_{*1}.\left(\Diamond\left(e_2 \wedge s_{*2}.(\square_{\ge 2}s \le 0.8\frac{s_{*1} + s_{*2}}{2})\right)\right)\right)$$
We consider a uniformly sampled trace with a sampling rate of 1 second. Lines are shown in order as in how the algorithm runs. For non-uniformly sampled traces, the steps are similar with one small difference: when to apply the trim operator when calculating $\text{intvl}(\psi)$ for any subformula ψ, this was explained in detail in section V-F.

In this example, our signal has just one component and it has the following values: $s = (3, 5, 8, 10, 14, 12, 11, 6, 3, 1, 7)$. The algorithm's first step is to calculate $\text{intvl}(\varphi_3) = [3, 3]$ and $\text{intvl}(\varphi_7) = [6, 6]$ (here, we assume e_1 is only TRUE at τ_3 and e_2 at τ_6) and $\text{sorted}(\varphi_{10}) = (1, 3, 3, 5, 6, 7, 8, 10, 11, 12, 14)$.

The algorithm starts with the first environment $\mathcal{E}[s_{*1} := 3, s_{*2} := 3]$, computes $\text{point}(\varphi_{10}) = [F, F, F, F, F, F, F, F, F, T, F]$ (these values are obtained by checking the condition $s \le 0.4(3 + 3)$ over the different signal values), $\text{flip}_{10}[0] = 1$ (this corresponds to the index of the value 3 in $\text{sorted}(\varphi_{10})$, 3 is the lowest s value that does not satisfy $s \le 0.4(3 + 3)$) and $\text{intvl}(\varphi_{10}) = [9, 9]$. Then the algorithm calls `ComputeIntervals` to calculate $\text{intvl}(\varphi_9) = \emptyset$. After that, it assigns the value F to $\text{point}(\varphi_8)[0]$ which represents the satisfaction relation $(i, \mathcal{E}[s_{*1} := 3, s_{*2} := 3]) \models \varphi_8$.

Then, the algorithm proceeds to the second environment $\mathcal{E}[s_{*1} := 3, s_{*2} := 5]$, $\text{flip}_{10}[1] = 3$ (now, the lowest s value that does not satisfy $s \le 0.4(3 + 5)$ is 5). The algorithm should update the value of $\text{point}(\varphi_{10})[0]$ and $\text{point}(\varphi_{10})[8]$ from F to T corresponding to the signal values with positions 1 and 2 in $\text{sorted}(\varphi_{10})$, however, since we already have the value of $\text{point}(\varphi_8)[0]$, we no longer need to look at or update $\text{point}(\varphi_{10})[0]$ and we can just skip it (more precisely, when freezing s_{*2} to σ_i, we only update $\text{point}(\varphi_{10})[k], k \ge i$). We obtain $\text{point}(\varphi_{10}) = [-, F, F, F, F, F, F, F, T, T, F]$ and $\text{intvl}(\varphi_{10}) = [8, 9]$. After that, the algorithm calls `ComputeIntervals` to compute $\text{intvl}(\varphi_9)$ then calculates $\text{point}(\varphi_8)[1]$.

The algorithm keeps on repeating the previous steps for all the environments $\mathcal{E}[s_{*1} := 3, s_{*2} := \sigma_i], 2 \le i \le |\pi| - 1$ which will result in computing all the values of $\text{point}(\varphi_8)$. With these values, the algorithm is able to compute $\text{intvl}(\varphi_8)$, $\text{intvl}(\varphi_6)$, $\text{intvl}(\varphi_5)$ and $\text{point}(\varphi_4)[0]$ for the environment group that freezes s_{*1} to 3.

Similarly, the algorithm will repeat all the previous steps for the environment group that freezes s_{*1} to $\sigma_i, 1 \leq i \leq |\pi| - 1$ in order to compute all point(φ_4) values.

Finally, the algorithm calculates intvl(φ_2) and intvl(φ_1) (Lines 5-7 from algorithm 1).

VII. QUANTITATIVE ROBUSTNESS FOR STL*

In this section, we define the quantitative semantics for STL* via a *robustness function* ρ which gives a measure of how well a trace satisfies or violates a given formula. As in the Boolean semantics, to reduce notation clutter, we will ommit π and simply write $\rho(\varphi, i, \mathcal{E})$ instead of $\rho(\varphi, \pi, i, \mathcal{E})$.

Definition 4 (Quantitative Semantics). Let $\pi = (\sigma_0, \tau_0), \ldots, (\sigma_{|\pi|-1}, \tau_{|\pi|-1})$ be a finite timed signal of arity dim, $\bowtie_> \in \{>, \geq\}$ and $\bowtie_< \in \{<, \leq\}$. For a given environment $\mathcal{E}$, and a given index position $0 \leq i \leq |\pi| - 1$, the robustness function valuation $\rho(\varphi, \pi, i, \mathcal{E}) \in \mathbb{R}$ for a STL* formula φ is defined as:

- $\rho(f(\overline{s}) \bowtie_> 0), i, \mathcal{E}) = f(\overline{\sigma_i})$.
- $\rho(f(\overline{s}) \bowtie_< 0), i, \mathcal{E}) = -f(\overline{\sigma_i})$.
- $\rho(f_1(\overline{s}) \bowtie_> f_2(\overline{s_*}), i, \mathcal{E}) = f_1(\overline{\sigma_i}) - f_2(\mathcal{E}(s_*^1), \ldots, \mathcal{E}(s_*^{\dim}))$.
- $\rho(f_1(\overline{s}) \bowtie_< f_2(\overline{s_*}), i, \mathcal{E}) = f_2(\mathcal{E}(s_*^1), \ldots, \mathcal{E}(s_*^{\dim})) - f_1(\overline{\sigma_i})$.
- $\rho(\neg \varphi, i, \mathcal{E}) = -\rho(\varphi, i, \mathcal{E})$.
- $\rho(\varphi_1 \wedge \varphi_2, i, \mathcal{E}) = \min(\rho(\varphi_1, i, \mathcal{E}), \rho(\varphi_2, i, \mathcal{E}))$.
- $\rho(\varphi_1 \vee \varphi_2, i, \mathcal{E}) = \max(\rho(\varphi_1, i, \mathcal{E}), \rho(\varphi_2, i, \mathcal{E}))$.
- $\rho(\Box_I \varphi, i, \mathcal{E}) = \min_{\tau_j \in \tau_i + I} (\rho(\varphi, j, \mathcal{E}))$.
- $\rho(\Diamond_I \varphi, i, \mathcal{E}) = \max_{\tau_j \in \tau_i + I} (\rho(\varphi, j, \mathcal{E}))$.
- $\rho(\varphi_1 \mathcal{U}_I \varphi_2, i, \mathcal{E}) = \max_{\tau_j \in \tau_i + I} \min \left(\rho(\varphi_2, j, \mathcal{E}), \min_{\tau_k \in [\tau_i, \tau_j)} \rho(\varphi_1, k, \mathcal{E}) \right)$.
- $\rho(s_*^k . \varphi, i, \mathcal{E}) = \rho(\varphi, i, \mathcal{E}[s_*^k := \sigma_i^k])$. $\quad\square$

Theorem 1. *Let φ be an STL* formula, i a position index, and $\mathcal{E}$ an environment. We have (i) if $\rho(\varphi, i, \mathcal{E}) > 0$ then$(i, \mathcal{E}) \models \varphi$; and (ii) if $\rho(\varphi, i, \mathcal{E}) < 0$ then $(i, \mathcal{E}) \not\models \varphi$. If $\rho(\varphi, i, \mathcal{E}) = 0$, nothing can be concluded.* $\quad\square$

A. STL* Quantitative Robustness Monitoring

To solve the robustness computation monitoring problem, we first examine the *decision problem*: is the robustness of a formula for a given trace greater than a given value r?

Theorem 2 (Robustness Decision Problem). *Let φ be an STL* formula, i a position index, $\mathcal{E}$ an environment and r a real number. We have (i) if $(i, \mathcal{E}) \models \varphi'$ then $\rho(\varphi, i, \mathcal{E}) \geq r$; and (ii) if $(i, \mathcal{E}) \not\models \varphi'$ then $\rho(\varphi, i, \mathcal{E}) \leq r$; where φ' is a logically equivalent syntactic transformation of φ obtained as follows: first negation symbols are removed from φ by pushing in negation and reversing any signal constraint or predicate if necessary; then we replace each of the below subformulae with its corresponding subformula (where $\bowtie_> \in \{>, \geq\}$ and $\bowtie_< \in \{<, \leq\}$).*

- *$f(\overline{s}) \bowtie_> 0$ replaced by $f(\overline{s}) \bowtie_> r$.*
- *$f(\overline{s}) \bowtie_< 0$ replaced by $f(\overline{s}) \bowtie_< -r$.*
- *$f_1(\overline{s}) \bowtie_> f_2(\overline{s_*})$ replaced by $f_1(\overline{s}) \bowtie_> f_2(\overline{s_*}) + r$.*
- *$f_1(\overline{s}) \bowtie_< f_2(\overline{s_*})$ replaced by $f_1(\overline{s}) \bowtie_< f_2(\overline{s_*}) - r$.* $\quad\square$

Now, given an STL* formula and a trace, we find an interval $[a, b]$ for which we are certain that the robustness valuation is within that interval.

Lemma 1. *Given a trace π, a position index $i \in [0, |\pi| - 1]$, an environment $\mathcal{E}$ and a negation-free STL* formula φ, for any subformula ψ in φ we have $\rho(\psi, \pi, i, \mathcal{E}) \in [a, b]$, where a and b are the lowest and highest robustness values signal predicates and constraints in φ can take over π.* $\quad\square$

Lemma 1 ensures computation of a conservative range of the robustness value before we run a monitoring algorithm. We note that any STL* formula can be transformed to a negation-free formula as demonstrated in theorem 2.

Example 8. Consider φ_1 from example 2. Suppose the highest value of s is 20 and the lowest is 5, then, we have $\rho(\varphi_1, 0, \epsilon) \in [-20 + 1.2 \times 5, 1.2 \times 20 - 5]$. In this example, we give the interval $[a, b]$ just by knowing sup and inf of s since f_1 and f_2 in the signal constraint $s \in [0.8 \frac{s_{*1} + s_{*2}}{2}, 1.2 \frac{s_{*1} + s_{*2}}{2}]$ are monotonic. $\quad\square$

Robustness Value Computation. Combining results from Theorem 2, Lemma 1 and the algorithm from Section V, we come up with a solution to the quantitative monitoring problem for STL*. Given an STL* formula φ and a trace π, we first use Lemma 1 to come up with a conservative range $[a, b]$ for $\rho(\varphi, 0, \epsilon)$. Next, we do a binary search over the interval $[a, b]$ employing our robustness decision problem monitoring algorithm over different formulae obtained as described in Theorem 2. The different formulae we will use are obtained by picking different r values from Theorem 2. With each new call of the monitoring algorithm, r will be the midpoint of the latest (which is also the smallest so far) conservative range of the robustness value. We continue the binary search until we reach the desired error range for the robustness value.

VIII. RUNNING TIME

A. STL* Boolean Monitoring Algorithm

Before giving the complexities of our algorithms, we introduce the following variables:

- $|\text{SubTree}(x_{*k})|$: # of sub-formulae in SubTree(x_{*k}).
- $|V|$: # of freeze variables in φ.
- $|\text{intvl}(\varphi)|$: maximal number of intervals for any φ_j in φ.
- $|\varphi|$: # of subformulae in φ.

We have the following complexities:

- Sorting a signal constraint takes $O(|\pi| \cdot \log(|\pi|))$.
- Compress Algorithm: $O(|\pi|)$.
- UpdateSignalConstraint Algorithm: $O(|\pi|)$.
- ComputeIntervals Algorithm: $O(|\text{intvl}(\varphi)|)$ for a uniformly sampled trace and $O\left(\min(|\pi|, |\text{intvl}(\varphi)| \cdot \log(|\pi|))\right)$ for non uniformly sampled trace.

For the Boolean monitoring algorithm, the complexity of the algorithm is the complexity of the recursive algorithm Rec-STL*(k, t). For a given call of Rec-STL* (for a given k and t values), the complexity of Lines 2-6 in Rec-STL* is $O(|\pi|)$ (we assume we have a constant number of signal constraints) and the complexity of Line 8 for loop is $O(|\varphi| \cdot |\text{intvl}(\varphi)|)$ for a uniformly sampled trace and $O(|\varphi| \cdot$

$\min\big(|\pi|, |\operatorname{intvl}(\varphi)| \cdot \log(|\pi|)\big)\big)$ for non uniformly sampled trace. And $\mathtt{Rec\text{-}STL}^*(k,t)$ is called $|\pi|^{|V|}$ times. Thus, the complexity of the Boolean monitoring algorithm is:

- $O\big(|\pi|^{|V|} \cdot \max(|\pi|, |\varphi|\cdot|\operatorname{intvl}(\varphi)|)\big)$ for uniformly sampled traces.
- $O\big(|\pi|^{|V|} \cdot \max\big(|\pi|, |\varphi|\cdot\min\big(|\pi|, |\operatorname{intvl}(\varphi)|\cdot\log(|\pi|)\big)\big)\big)$ for non-uniform traces.

In practice, as we mentioned earlier, we expect $|\operatorname{intvl}(\varphi)|$ to be much smaller compared to trace size $|\pi|$.

If we drop the intervals idea and data structure, we obtain a *non-interval* algorithm inspired by MTL monitoring algorithms, tweaked so that it can be used in the context of STL*. The time complexity of this algorithm is $O\big(c\cdot|\varphi|\cdot|\pi|^{|V|+1}\big)$ (with $c = \lceil a/\Delta \rceil$ where a is the largest constant occurring in the temporal operators in φ, and Δ is the smallest difference between two consecutive timestamps, in worst case, $c = |\pi|$). For a given environment and a given subformula, this non-interval algorithm has to compute all satisfaction relations for the different trace points while our algorithm can skip points using the intervals data structure. Additionally, the non-interval algorithm uses recursive formulae for temporal operators [31] which is why we see the new c factor in its complexity (for a given subformula with a timed temporal operator, computing a single satisfaction relation for a given index position and a given environment takes $O(c)$ compared to our algorithm where we avoid it using the intervals data structure. Note here, if one is not careful and does not use the recursive formula for the timed temporal operators, we end up with complexity $O\big(|\varphi|\cdot|\pi|^{|V|+2}\big)$.

B. STL* Quantitative Robustness Computation Algorithm

Proposition 1. *Given an* STL* *formula* φ *and an error value* $\epsilon \in \mathbb{R}_+$, *it takes* $O(|\pi|^{|V|})$ *time to obtain an initial conservative range* $[a, b]$ *of* $\rho(\varphi, 0, \mathcal{E})$ *and* $n = \lceil \log_2(\frac{b-a}{\epsilon}) \rceil$ *calls of the monitoring algorithm to obtain a conservative range with a width* $\le e$. *That range represents our estimation for* $\rho(\varphi, 0, \mathcal{E})$. $\square$

We conclude the following complexities for our STL* robustness algorithm is:

- $O\big(n \cdot |\pi|^{|V|} \cdot \max(|\pi|, |\varphi|\cdot|\operatorname{intvl}(\varphi)|)\big)$ for uniformly sampled traces.
- $O\big(n \cdot |\pi|^{|V|} \cdot \max\big(|\pi|, |\varphi|\cdot\min(|\pi|, |\operatorname{intvl}(\varphi)|\cdot\log(|\pi|))\big)\big)$ for non-uniform traces.

We recall the complexity of the algorithm from [21] is $O(|\varphi|\cdot |\pi|^{|V|+2})$.

IX. EXPERIMENTS

We conducted our experiments on a 64-bit Intel(R) i7-12700H @ 2.30 GHz with 32-GB RAM and we implemented our algorithms using C++. We tested our algorithms on the formulae $\varphi_1 \ldots \varphi_4$ from section III:

- $\varphi_1 = \Diamond\Big(e_1 \wedge s_{*1}.\Big(\Diamond\big(e_2 \wedge s_{*2}.\Box_{[2,T]}s \in [0.8\frac{s_{*1}+s_{*2}}{2}, 1.2\frac{s_{*1}+s_{*2}}{2}]\big)\Big)\Big)$
- $\varphi_2 = \Diamond\Big(s^1 > 5 \wedge s_{*1}^2.\Diamond\big(s^1 > 10 \wedge s_{*2}^2.\Diamond\big((s^2 > s_{*1}^2 + s_{*2}^2)\mathcal{U}s^1 < 5\big)\big)\Big)$
- $\varphi_3 = \Box s_{*1}.\big(|s_{*1}-s| \le \epsilon \, \mathcal{U}\big(|s_{*1}-s| \ge \Delta \wedge s_{*2}.(|s_{*2}-s| \le \epsilon \, \mathcal{U}|s_{*1}-s| \le \epsilon)\big)\big)$
- $\varphi_4 = \Box s_{*1}.\Big(|s_{*1}-s| \le \epsilon \, \mathcal{U}\Big(|s_{*1}-s| \ge \Delta \wedge s_{*2}.\Big(|s_{*2}-s| \le \epsilon$

$\mathcal{U}\big(|s_{*2}-s| \ge \Delta \wedge \, s_{*3}.(|s_{*3}-s| \le \epsilon \, \mathcal{U} \, |s_{*1}-s| \le \epsilon)\big)\Big)\Big)\Big)$

We generated the traces using Python. Trace noise added by superimposing a noise signal. For each of the 4 formulae above, we picked two relevant traces, one of which satisfied the requirement (π_s) and the other of which violated it (π_v). For example, for φ_3, we used traces made of the signal shown in Figure 1. We ran each of the formulae on different trace sizes: the same signal over a constant time horizon was sampled with different sampling rates to vary the trace size.

A. STL* Boolean Monitoring Algorithms

The traces used in Tables I are uniformly sampled while the traces in Tables II are obtained by having random sampling points, in other words, we do not use a fixed sampling rate and the timestamps are randomly selected. We additionally equip our algorithm with an early stoppage condition for formulae starting with $\Box$ or $\Diamond$ (for $\Box\psi$, if ψ is FALSE once, we return FALSE. Similarly if ψ is TRUE for $\Diamond\psi$).

φ	$|\operatorname{intvl}|$	$|\pi|=500$		$|\pi|=1k$		$|\pi|=2k$		$|\pi|=4k$		$|\pi|=10k$	
		π_s	π_v	π_s	π_v	π_s	π_v	π_s	π_v	π_s	π_v
φ_1	5	0.04	0.36	0.22	1.47	0.96	5.88	3.86	24	24.2	151
φ_2	6	0.04	0.37	0.18	1.52	0.72	6.06	2.88	24	18.2	152
φ_3	11	0.45	0.02	1.84	0.11	7.31	0.44	29	1.76	183	11
φ_4	22	157	14.2	22m	2m	178m	17m	-	-	-	-

TABLE I: Running times in seconds (or minutes, m) of STL* Boolean monitoring algorithm over uniformly sampled traces. Trace π_s satisfies the formulae and π_v violates it.

φ	$|\operatorname{intvl}|$	$|\pi|=500$		$|\pi|=1k$		$|\pi|=2k$		$|\pi|=4k$		$|\pi|=10k$	
		π_s	π_v	π_s	π_v	π_s	π_v	π_s	π_v	π_s	π_v
φ_1	5	0.04	0.36	0.22	1.46	0.96	5.90	3.85	24	24.2	151
φ_2	6	0.04	0.38	0.19	1.53	0.73	6.10	2.89	24	18.3	153
φ_3	11	0.45	0.02	1.87	0.12	7.45	0.47	30	1.78	184	11.1
φ_4	22	158	14.3	23m	2m	184m	18m	-	-	-	-

TABLE II: Running times of STL* Boolean monitoring algorithm over non-uniformly sampled traces.

We also implemented a Boolean version of the STL* robustness monitoring algorithm from [21] and equipped it with the same early stoppage condition to make the comparison fair. We report the running times in Table III. Note that that our formulae do not have timed until operators, that is why the complexity of the non-interval algorithm from [21] in this case is $O\big(|\varphi|\cdot|\pi|^{|V|+1}\big)$. In the general case, the complexity would be $O\big(|\varphi|\cdot|\pi|^{|V|+2}\big)$.

φ	$|\pi|=500$		$|\pi|=1k$		$|\pi|=2k$		$|\pi|=4k$		$|\pi|=10k$	
	π_s	π_v	π_s	π_v	π_s	π_v	π_s	π_v	π_s	π_v
φ_1	0.21	1.84	2.07	14.8	16.9	119	121	16m	37m	262m
φ_2	0.18	1.80	1.52	14.4	14.7	116	114	16m	31m	255m
φ_3	1.82	0.11	14.6	0.88	118	7.12	16m	56	255m	16m
φ_4	462	41.7	124m	11m	-	-	-	-	-	-

TABLE III: Running times in seconds (or minutes, m) of non-interval STL* Boolean monitoring algorithm.

Our experimental results show that our algorithm outperforms the non-interval algorithm. We can see that, in practice, the number of intervals $|\operatorname{intvl}(\varphi_i)|$ is much smaller compared to $|\pi|$. In addition, $|\operatorname{intvl}(\varphi_i)|$ is independent of the trace size which is expected since we use same signals with the same time horizon, with different sampling rates. We also notice that the early stoppage condition helps reduce the running times significantly.

The formulae φ_1, φ_2 and φ_3 have 2 freeze variables and we can see, for the cases where there is no early stoppage, an almost quadratic running time in proportion with the trace size, however, when we look at the previous complexity analysis, it indicates a cubic dependence. This can be explained as follows: `UpdateSignalConstraint` often does not require $O(|\pi|)$ running time in practice: from one instantiation to the next one, only few values, and not all $|\pi|$ values, in $\text{point}(\psi)$ (where ψ is the signal constraint) will need updates while the majority of values will remain the same. This is due to the fact that signals in real-world systems are continuous, and in our case for pointwise semantics, from one timestamp to the next one, we do not expect large trace value change. Thus, from one environment to the next, the values of the f_2 function in the signal constraints do not have sudden shifts (for example, if we had non-continuous f_2 functions in our signal constraints, we would expect a higher number of values needs to be updated). Hence, we expect `UpdateSignalConstraint` to run in $O(\log(|\pi|))$, the time needed to find the new value of flip_j using $\text{sorted}(\varphi_j)$. With that assumption, the complexity can be simplified to

- $O\Big(|\pi|^{|V|}\cdot\max\big(\log(|\pi|), |\varphi|.|\,\text{intvl}(\varphi)|\big)\Big)$ for a uniformly sampled trace.
- $O\left(|\pi|^{|V|}\cdot\max\left(\begin{array}{l}\log(|\pi|),\\ |\varphi|\cdot\min(|\pi|, |\,\text{intvl}(\varphi)|\cdot\log(|\pi|))\end{array}\right)\right)$ or simply $O\Big(|\pi|^{|V|}\cdot|\varphi|\cdot\min\big(|\pi|, |\,\text{intvl}(\varphi)|\cdot\log(|\pi|)\big)\Big)$ for a non-uniformly sampled trace.

For φ_4 which has 3 freeze variables, our algorithm starts running slow once the trace gets larger, which is to be expected from our complexity analysis.

B. STL* *Quantitative Robustness Computation Algorithms*

For computing robustness, we implemented two algorithms: (i) a non-interval STL* robustness monitoring algorithm from [21] (results in Table IV); and (ii) the interval STL* robustness monitoring algorithm from Subsection VII-A (results in Table V). The experiments were run on uniformly sampled traces; times for non-uniformly sampled traces are expected to be similar based on Table I and Table II results.

φ	$|\pi| = 500$	$|\pi| = 1k$	$|\pi| = 2k$	$|\pi| = 4k$	$|\pi| = 10k$
φ_1	1.91	15.31	123	17m	266m
φ_2	1.83	14.65	118	16m	252m
φ_3	1.94	15.52	124	17m	267m
φ_4	475	126m	-	-	-

TABLE IV: Running times in seconds (or minutes, m) of non-interval STL* robustness monitoring algorithm.

Non-interval STL* robustness monitoring algorithm [21]. This algorithm has the following complexity $O\left(|\varphi|\cdot|\pi|^{|V|+2}\right)$. It uses the $\max/\min$ filter from [32] to reduce the complexity to $O\left(|\varphi|\cdot|\pi|^{|V|+1}\right)$ for formulae without timed until operator. We recall that in our experiments, all formulae do not have timed until. We note that this algorithm cannot have the early stoppage condition for $\Box$ and $\Diamond$ since this option can only be applied for the Boolean case. The experiment results are in Table IV.

Our interval STL* robustness monitoring algorithm from Subsection VII-A stops the binary search once it reaches a

conservative range $e \leq 0.1$ for the robustness. In Table V, we also report the relative error of the estimated robustness value compared to the exact value.

φ	$|\text{intvl}|$	n	i.c.r.w.	r.e	$|\pi| = 500$	$|\pi| = 1k$	$|\pi| = 2k$	$|\pi| = 4k$	$|\pi| = 10k$
φ_1	8	9	49	1%	2.16	9.02	38.1	149	15m
φ_2	8	10	83	$\leq 1\%$	2.31	9.38	39.5	154	16m
φ_3	13	9	43	2%	2.47	10.3	41	166	17m
φ_4	25	9	173	$\leq 1\%$	15m	126m	-	-	-

n: number of times the STL* Boolean algorithm is called.
i.c.r.w: initial conservative range width, $b - a$ in proposition 1.
r.e: estimated robustness value relative error.

TABLE V: Running times in seconds (or minutes, m) of interval STL* robustness monitoring algorithm.

The obtained n values in Table V confirm our analysis in proposition 1. We can also see that the values for $\text{intvl}(\varphi)$ are slightly larger compared to the previous experiments, this can be explained by the fact that we are running the Boolean monitoring algorithm over slightly different formulae and not the original formulae $\varphi_1 \ldots \varphi_4$. The new modified formulae give us in some cases lower $\text{intvl}(\varphi)$ value and in other cases higher value. The running times conform with our complexity analysis and show that, in most cases, the accelerated STL* robustness computation algorithm scales better than the non-interval one.

X. CONCLUSION

In this work we presented an acceleration heuristic using intervals for monitoring STL* specifications. We showed that with this heuristic, monitoring for STL* specifications with two nested freeze variables remains tractable for Boolean monitoring as well as for robustness monitoring; and somewhat tractable for three nested freeze variables. We posit engineering properties of interest that can be expressed in STL* can be expressed in the STL* subset of two, and occasionally three, nested freeze variables. Ours is the *first* work which presents implemented Boolean and robustness monitoring algorithms for formulae with nested freeze quantifiers. For the robustness value computation, we first presented algorithms for the corresponding decision problem using the acceleration heuristic, and then computed the robustness value using binary search. A notable feature of using the decision problem procedure for the robustness value computation is that it allows early stoppage for $\Box$ and $\Diamond$ operators; such early stoppage is not possible in a direct robustness value computation which does not use the decision problem algorithm.

One of the main applications of temporal logic robustness is in the test-generation setting where black-box optimizers are used to search for an input such that the corresponding system output robustness value is negative, falsifying the logical specification [33], [34], [35], [36], [37], [12], [38]. In such a setting, one could conceivably stop the robustness binary searches earlier – and thus gain even more in terms of time – if the robustness value range is positive and greater than the robustness values seen for previous inputs, as we aim to search for an input that drives the robustness value lower than those seen so far. How this would impact the falsification process depends on the optimizer used and how it uses the actual robustness values. We plan to investigate this line of research in follow-up work.

REFERENCES

[1] G. E. Fainekos, S. Sankaranarayanan, K. Ueda, and H. Yazarel, "Verification of automotive control applications using s-taliro," in *ACC*, pp. 3567–3572, IEEE, 2012.

[2] V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia, "Reactive synthesis from signal temporal logic specifications," in *HSCC*, pp. 239–248, ACM, 2015.

[3] V. Raman, A. Donzé, M. Maasoumy, R. M. Murray, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, "Model predictive control for signal temporal logic specification," *CoRR*, vol. abs/1703.09563, 2017.

[4] J. V. Deshmukh, M. Horvat, X. Jin, R. Majumdar, and V. S. Prabhu, "Testing cyber-physical systems through bayesian optimization," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 170:1–170:18, 2017.

[5] S. Sankaranarayanan, S. A. Kumar, F. Cameron, B. W. Bequette, G. Fainekos, and D. M. Maahs, "Model-based falsification of an artificial pancreas control system," *SIGBED Rev.*, vol. 14, no. 2, pp. 24–33, 2017.

[6] E. Bartocci, J. V. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Nickovic, and S. Sankaranarayanan, "Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications," in *Lectures on Runtime Verification - Introductory and Advanced Topics*, vol. 10457 of *LNCS*, pp. 135–175, Springer, 2018.

[7] Z. Kong, A. Jones, and C. Belta, "Temporal logics for learning and detection of anomalous behavior," *IEEE Trans. Autom. Control.*, vol. 62, no. 3, pp. 1210–1222, 2017.

[8] G. Ernst, S. Sedwards, Z. Zhang, and I. Hasuo, "Falsification of hybrid systems using adaptive probabilistic search," *ACM Trans. Model. Comput. Simul.*, vol. 31, no. 3, pp. 18:1–18:22, 2021.

[9] W. Liu, N. Mehdipour, and C. Belta, "Recurrent neural network controllers for signal temporal logic specifications subject to safety constraints," *IEEE Control. Syst. Lett.*, vol. 6, pp. 91–96, 2022.

[10] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Syst.*, vol. 2, no. 4, pp. 255–299, 1990.

[11] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia, "Robust online monitoring of signal temporal logic," *Formal Methods Syst. Des.*, vol. 51, no. 1, pp. 5–30, 2017.

[12] M. Waga, "Falsification of cyber-physical systems with robustness-guided black-box checking," in *HSCC'20*, pp. 11:1–11:13, ACM, 2020.

[13] L. Brim, P. Dluhos, D. Safránek, and T. Vejpustek, "Stl*: Extending signal temporal logic with signal-value freezing operator," *Inf. Comput.*, vol. 236, pp. 52–67, 2014.

[14] A. Bakhirkin and N. Basset, "Specification and efficient monitoring beyond STL," in *TACAS*, vol. 11428 of *LNCS*, pp. 79–97, Springer, 2019.

[15] S. Silvetti, L. Nenzi, E. Bartocci, and L. Bortolussi, "Signal convolution logic," in *ATVA*, vol. 11138 of *LNCS*, pp. 267–283, Springer, 2018.

[16] R. Alur and T. A. Henzinger, "A really temporal logic," *J. ACM*, vol. 41, no. 1, pp. 181–204, 1994.

[17] P. Bouyer, F. Chevalier, and N. Markey, "On the expressiveness of TPTL and MTL," *Inf. Comput.*, vol. 208, no. 2, pp. 97–116, 2010.

[18] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu, "Monitoring metric first-order temporal properties," *J. ACM*, vol. 62, may 2015.

[19] A. Bakhirkin, T. Ferrère, T. A. Henzinger, and D. Nickovic, "The first-order logic of signals: keynote," in *EMSOFT 2018*, p. 1, IEEE, 2018.

[20] A. Donzé, T. Ferrère, and O. Maler, "Efficient robust monitoring for STL," in *CAV'13*, LNCS 8044, pp. 264–279, Springer, 2013.

[21] L. Brim, T. Vejpustek, D. Safránek, and J. Fabriková, "Robustness analysis for value-freezing signal temporal logic," *EPTCS*, vol. 125, pp. 20–36, 2013.

[22] B. Ghorbel and V. S. Prabhu, "Fast and scalable monitoring for value-freeze operator augmented signal temporal logic," in *HSCC '24*, ACM, 2024.

[23] B. Ghorbel and V. S. Prabhu, "Fast robust monitoring for signal temporal logic with value freezing operators (STL*)." arXiv 2408.02460, 2024.

[24] A. Donzé, O. Maler, E. Bartocci, D. Nickovic, R. Grosu, and S. A. Smolka, "On temporal logic and signal processing," in *ATVA 2012*, vol. 7561 of *LNCS*, pp. 92–106, Springer, 2012.

[25] L. V. Nguyen, J. Kapinski, X. Jin, J. V. Deshmukh, K. Butts, and T. T. Johnson, "Abnormal data classification using time-frequency temporal logic," in *HSCC'17*, p. 237242, ACM, 2017.

[26] C. Boufaied, M. Jukss, D. Bianculli, L. C. Briand, and Y. Isasi Parache, "Signal-based properties of cyber-physical systems: Taxonomy and logic-based characterization," *Journal of Systems and Software*, vol. 174, p. 110881, 2021.

[27] A. Dokhanchi, B. Hoxha, C. E. Tuncali, and G. Fainekos, "An efficient algorithm for monitoring practical TPTL specifications," in *MEM-OCODE*, pp. 184–193, IEEE, 2016.

[28] B. Ghorbel and V. S. Prabhu, "Linear time monitoring for one variable TPTL," in *HSCC '22*, pp. 5:1–5:11, ACM, 2022.

[29] B. Ghorbel and V. S. Prabhu, "Quantitative robustness for signal temporal logic with time-freeze quantifiers," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 42, no. 12, pp. 4436–4449, 2023.

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

[31] P. Thati and G. Rosu, "Monitoring algorithms for metric temporal logic specifications," *ENTCS*, vol. 113, pp. 145–162, 2005. RV'04.

[32] D. Lemire, "Streaming maximum-minimum filter using no more than three comparisons per element," *Nord. J. Comput.*, vol. 13, no. 4, pp. 328–339, 2006.

[33] T. Akazaki and I. Hasuo, "Time robustness in MTL and expressivity in hybrid system falsification," in *CAV 2015, Proceedings, Part II*, vol. 9207 of *LNCS*, pp. 356–374, Springer, 2015.

[34] Y. Annpureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan, "S-taliro: A tool for temporal logic falsification for hybrid systems," in *TACAS 2011*, vol. 6605 of *LNCS*, pp. 254–257, Springer, 2011.

[35] B. Barbot, N. Basset, T. Dang, A. Donzé, J. Kapinski, and T. Yamaguchi, "Falsification of cyber-physical systems with constrained signal spaces," in *NFM 2020,*, vol. 12229 of *LNCS*, pp. 420–439, Springer, 2020.

[36] A. Dokhanchi, S. Yaghoubi, B. Hoxha, and G. Fainekos, "Vacuity aware falsification for MTL request-response specifications," in *CASE 2017*, pp. 1332–1337, IEEE, 2017.

[37] Z. Ramezani, A. Donzé, M. Fabian, and K. Åkesson, "Temporal logic falsification of cyber-physical systems using input pulse generators," in *ARCH 2021*, vol. 80 of *EPiC Series in Computing*, pp. 195–202, EasyChair, 2021.

[38] Z. Zhang, P. Arcaini, and I. Hasuo, "Constraining counterexamples in hybrid system falsification: Penalty-based approaches," in *NFM 2020*, vol. 12229 of *LNCS*, pp. 401–419, Springer, 2020.