Fast and Scalable Monitoring for Value-Freeze Operator augmented Signal Temporal Logic

Bassem Ghorbel
Colorado State University
USA
bassemghorbel99@gmail.com

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

ABSTRACT

Signal Temporal Logic (STL) is a timed temporal logic formalism that has found widespread adoption for rigorous specification of properties in Cyber-Physical Systems. However, STL is unable to specify oscillatory properties commonly required in engineering design. This limitation can be overcome by the addition of additional operators, for example, signal-value freeze operators, or with first order quantification. Previous work on augmenting STL with such operators has resulted in intractable monitoring algorithms. We present the first efficient and scalable offline monitoring algorithms for STL augmented with independent freeze quantifiers. Our final optimized algorithm has a $|\rho| \log(|\rho|)$ dependence on the trace length $|\rho|$ for most traces ρ arising in practice, and a $|\rho|^2$ dependence in the worst case. We also provide experimental validation of our algorithms – we show the algorithms scale to traces having 100k time samples.

ACM Reference Format:
Bassem Ghorbel and Vinayak S. Prabhu. 2024. Fast and Scalable Monitoring for Value-Freeze Operator augmented Signal Temporal Logic. In *27th ACM International Conference on Hybrid Systems: Computation and Control (HSCC '24), May 14–16, 2024, Hong Kong SAR, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3641513.3650128

1 INTRODUCTION

A core requirement of rigorous system design is a mechanism for precisely specification and identification of what are good and what are bad executions. Automata and temporal logics, and corresponding monitoring algorithms for executions are two such commonly used frameworks [1, 3, 20, 25, 26, 30, 32, 33]. In the context of Cyber-Physical Systems (CPS) and Control, *Signal Temporal Logic* (STL) has found wide adoption as a specification formalism [6, 14, 17, 22, 23, 27–29, 31]. Contributing factors to its widespread adoption are: (i) its expressivity, allowing precise characterizations of complex timed requirements, (ii) a mechanism to translate quantitative signal values to Boolean predicates allowing a move to the timed temporal logic MTL (Metric Temporal Logic) [25], (iii) fast monitoring algorithms for checking when an execution satisfies an STL specification [10, 13]; and (iv) a corresponding robustness function which quantifies how well an execution satisfies or violates the given STL specification [16].

The mechanism by which STL translates quantitative signal values to Boolean predicates – comparing signal values to constant thresholds – has limitations. For example, STL is unable to specify oscillatory properties, an important property of interest in biological and engineering systems. This limitation has been noted by researchers, and STL augmentations with *freeze* or *first order* quantification have been proposed in order to overcome expressivity limitations [5, 9]. Both augmentations allow capture of signal values to be used for comparison with later trace values. Captured signal values can then be used in constraints such as (example taken from [9] which presents the logic STL* containing value freezing quantification) "there repeatedly occur time instants such that for the corresponding signal value v at such time instants, there are two near futures (within at most 10s), when the signal value is $\leq v + \delta$, also $\geq v + \delta$ respectively". Such a constraint specifies oscillations of amplitude at least δ, and it is believed it cannot be encoded in STL [9]. In this example, the trace values v are repeatedly captured in a single freeze variable over the duration of the trace.

However, increased expressivity has incurred a heavy algorithmic penalty for monitoring and related algorithms in previous work [5, 9]. Augmentation with signal value freezing operators was proposed in [9]; the monitoring algorithm is complex, and involves manipulation of polygons. Even in the case of a single freeze operator, and even for *approximate* monitoring in an attempt to make the problem tractable, the algorithm in [9] is very intricate. Due to the complicated nature of the algorithm which involves manipulating polygons, [9] did not obtain a precise complexity bound: it was only shown that "each of the steps of the algorithm has at most polynomial complexity to the number of polygons and the number of polygons grows at most polynomialy in each of the steps". Their monitoring experiments showed scalability limitations – over an hour of running time for signals containing 100 timepoints. First order quantification similarly leads to a complex algorithm. The work of [5] proposes a monitoring algorithm, and derives a running time bound of $2^{(|\varphi|+|\rho|)^{2^{O(f+l)}}}$, where φ is the formula in their logic, ρ is the trace, and f is the number of freeze quantifiers (for our discussion l is not relevant). The constant in the big O in the second exponent is not derived. They also show that for a subset of their logic, the length of the trace $|\rho|$ in the first exponent can be moved out, but still the big O remains in the second exponent. An implementation of the algorithms is not presented.

Thus, previous monitoring algorithms for both STL augmented with signal freeze quantifiers, and for the first order logic of signals, are prohibitively complicated, and stand in stark contrast to efficient

HSCC '24, May 14–16, 2024, Hong Kong SAR, China

ACM ISBN 979-8-4007-0522-9/24/05
https://doi.org/10.1145/3641513.3650128

monitoring algorithms for STL, which have been implemented in various tools such as Breach and S-TaLiRo [12, 15]. Efficient monitoring algorithms for MTL augmented with *time* freeze quantifiers have previously been presented in [11, 18, 19].

Our Contributions. In this work, we consider a fragment of STL*: one that augments STL with freeze quantifiers such that only one quantifier is "active" at a time; that is, all freeze variables are independent of each other. This fragment, which we denote as STL$^*_1$, subsumes STL, and is general enough to express all of the STL* properties, except for one, from [9]. We present *efficient* and *implementable* offline monitoring algorithms for this fragment. We consider two types of traces ρ with signal values in $\mathbb{R}^n$: uniformly time-sampled where the trace is sampled at times $0, \Delta, 2\Delta, 3\Delta, \ldots$, and non-uniformly time-sampled where the inter-sample time need not be a constant. For the uniformly sampled case, a naive algorithm would run in time $\Omega(|\rho|^{V+1})$ where $|\rho|$ is the number of trace samples, and V is the number of freeze variables, as we would need to look at all possible freeze capturings of the V freeze variables (contributing $|\rho|^V$) , and also iterate over the whole trace to compute formula satisfaction at different trace timepoints (contributing $|\rho|$). We first present a polynomial time algorithm (building upon the idea in [11]) which leverages independence of the freeze variables so that we need only consider freeze bindings of each freeze variable separately (and not in a nested fashion) and results in a running time of $O\left(c \cdot V \cdot |\varphi| \cdot |\rho|^2\right)$, where $c = \lceil a/\Delta \rceil$ where a is the largest constant occurring in the temporal operators in φ, and Δ is the period between two consecutive timestamps; removing the exponential dependence on V. Next, we present an optimized algorithm that removes the c factor, and also reduces the quadratic dependence on the trace length for practical traces. This optimization uses the observation that temporally close sample values in real world traces have close signal values, and hence formula satisfaction values are constant for most of the trace timepoints, with satisfaction value "toggles" from true to false, or vice-versa, occurring at far fewer points as compared to the trace length. The optimized algorithm uses work done when a freeze variable was bound to the i-th signal value when it works on the freeze variable binding to the $i+1$-th signal value sample. This results in $O\left(|\rho| \cdot \log(|\rho|) + V \cdot |\varphi| \cdot |\rho| \cdot |\text{intvl}\,\varphi|\right)$ running time in practice, where $|\text{intvl}\,\varphi|$ denotes the number of true/false toggles over the trace for any φ subformula. The number $|\text{intvl}\,\varphi|$ is much smaller than the trace length $|\rho|$ in practice, resulting in a $|\rho|\log(|\rho|)$ dependence on the trace length rather than $|\rho|^2$ for traces from real-world systems. The optimized algorithm over the interval data structure we use also removes the c factor. We also obtain corresponding algorithms for non-uniformly sampled traces.

Lastly, we present experimental validation for our optimized algorithm implemented in C++. For both the uniformly sampled and the non-uniformly sampled cases, our algorithms scale to trace lengths of 100k. The running times are under 2 minutes even for such large trace lengths. Our results can be contrasted with the approximate monitoring results from [9] where the running time was over an hour even for trace lengths of 100 samples (for the same STL$^*_1$ fragment we consider). While admittedly the framework of [9] is over piecewise linear continuous-time traces, and ours is over time-sampled traces, since we can only observe the real world at a sequence of timepoints, a piecewise linear interpretation is still an approximation of what happens in between observations. One can simply oversample the traces and use our scalable algorithms. We also note that our monitoring algorithms are *exact* monitoring algorithms for time sampled traces.

2 VALUE-FREEZING SIGNAL TEMPORAL LOGIC

Signals/Traces. A $\mathbb{R}^n$ valued *signal* or a *trace* is a pair (σ, τ), where $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_{|\rho|-1}$ is a finite sequence of elements from $\mathbb{R}^n$, and $\tau = \tau_0, \tau_1, \ldots, \tau_{|\rho|-1}$ are the corresponding timestamps from $\mathbb{R}_+$. The signal value at timestamp τ_i is $\sigma_i \in \mathbb{R}^n$ and i is a position index. The k-th signal dimension of $\sigma = \langle a_1, \ldots, a_n \rangle$, namely a_k, is denoted $\sigma(k) = \sigma_0(k), \sigma_1(k), \ldots, \sigma_{|\rho|-1}(k)$. In order to simplify the presentation, we sometimes assume a timed word to be of the type $(\sigma_0, \tau_0), \ldots (\sigma_{|\rho|-1}, \tau_{|\rho|-1})$. 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*; otherwise the traces are *non-uniformly sampled*.

Definition 1 (STL* Syntax)**.** Given a signal arity n, and a finite set of *freeze variables* $\{s_k^1, \ldots, s_k^m\}$ for each signal dimension $1 \le k \le n$, the formulae of value-freezing signal temporal logic (STL*) are defined by the grammar:

$$\varphi := s_k \sim r \mid s_k^{h*} \sim s_{k'} \pm r \mid \neg\varphi \mid \varphi_1 \underset{\wedge}{\vee} \varphi_2 \mid \Box_I \varphi \mid \Diamond_I \varphi \mid \varphi_1 \mathcal{U}_I \varphi_2 \mid s_k^h.\varphi$$

where s_k and $s_{k'} \in \{s_1, \ldots, s_n\}$ are *signal variables* (s_k refers to the k-th signal dimension), r is a positive real, and $I = [a, b]$ is an interval where a and b are positive reals, s_k^{h*} is a *frozen value* corresponding to signal-value freeze variables s_k^h for signal dimension k, and $\sim \in \{<, >, \le, \ge, =\}$ is the standard comparison operator. □

For ease of reference, we have $\Box$ and $\Diamond$ as explicit operators. The freeze operator "$s_k^h.$" binds the current value of the k-th signal dimension to the frozen value s_k^{h*}. Note that we allow the frozen value s_k^{h*} for the k-th signal signal dimension to be compared to a current signal value for a different signal dimension k' in $s_k^{h*} \sim s_{k'} \pm r$. We refer to $s_k \sim r$ as a *signal predicate*, and to $s_k^{h*} \sim s_{k'} \pm r$ as a *freeze signal constraint*.

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

- $(\rho, i, \mathcal{E}) \models s_k \sim r$ iff $\sigma_i(k) \sim r$ for signal variable s_k.
- $(\rho, i, \mathcal{E}) \models \neg\varphi$ iff $(\rho, i, \mathcal{E}) \not\models \varphi$.
- $(\rho, i, \mathcal{E}) \models \varphi_1 \underset{\wedge}{\vee} \varphi_2$ iff $(\rho, i, \mathcal{E}) \models \varphi_1 \underset{\text{and}}{\text{or}} (\rho, i, \mathcal{E}) \models \varphi_2$.
- $(\rho, i, \mathcal{E}) \models \Diamond_{[a,b]} \varphi$ iff $\exists j$, $\tau_j \in \tau_i + [a, b]$, s.t. $(\rho, j, \mathcal{E}) \models \varphi$.
- $(\rho, i, \mathcal{E}) \models \Box_{[a,b]} \varphi$ iff $\forall j$, $\tau_j \in \tau_i + [a, b]$, s.t. $(\rho, j, \mathcal{E}) \models \varphi$.
- $(\rho, i, \mathcal{E}) \models \varphi_1 \mathcal{U}_{[a,b]} \varphi_2$ iff $(\rho, j, \mathcal{E}) \models \varphi_2$ for some $j \ge i$ with $\tau_j \in \tau_i + [a, b]$ and $(\rho, k, \mathcal{E}) \models \varphi_1$ $\forall i \le k < j$.
- $(\rho, i, \mathcal{E}) \models s_k^h.\varphi$ iff $(\rho, i, \mathcal{E}[s_k^h := \sigma_i(k)]) \models \varphi$; where $\mathcal{E}[s_k^h := \sigma_i(k)]$ denotes the environment $\mathcal{E}'$ defined as $\mathcal{E}'(x) = \mathcal{E}(x)$ for $x \neq s_k^h$, and $\mathcal{E}'(s_k^h) = \sigma_i(k)$.

• $(\rho, i, \mathcal{E}) \models s_k^{h*} \sim s_{k'} \pm r$ iff $\mathcal{E}(s_k^h) \sim \sigma_i(k') \pm r$.

We say the trace ρ satisfies an STL* formula φ if $(\rho, 0, \mathcal{E}[\equiv\sigma_0]) \models \rho$ where $\mathcal{E}[\equiv\sigma_0]$ denotes the freeze variable environment where all variables s_k^h are mapped to their corresponding $\sigma_0(k)$ values. □

Given a signal of arity n, for the i-th timestamp τ_i, we refer to the k-th signal dimension value as $s_k(\tau_i)$, which has the value $\sigma_i(k)$. In order to reduce notation clutter, we will simply write $(i, \mathcal{E})$ instead of $(\rho, i, \mathcal{E})$ in the remainder of this paper since for any given STL* formula, we will be using the same trace ρ. We use the phrase i^{th} *instantiation* of a freeze variable s_k^h to mean the environment $\mathcal{E}$ where the freeze variable s_k^h is assigned the value $\sigma_i(k)$.

Note: We can freeze the same signal dimension multiple times in an STL* formula. The superscript $h \in \mathbb{N}_{>0}$ is used in that case to indicate which frozen value refers to which signal-value freeze operator. Thus, $s_k^{h_1}$ and $s_k^{h_2}$ for $h_1 \neq h_2$ are considered different freeze variables, but each freezing the k-th signal dimension at different times. When we freeze a signal dimension only once in a STL* formula, we omit the superscript $h = 1$.

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

A freeze variable s_k^h occuring in a frozen value context s_k^{h*} is said to be *free* if it is not in the scope of a corresponding freeze operator "$s_k^h.$". The set of free variables for a formula φ is denoted by $\text{Free}(\varphi)$ (the formal defintion is in the appendix). It can be shown that the environment $\mathcal{E}$ is only relevant for the free variables in an STL* formula when it comes to the satisfaction relation $(i, \mathcal{E}) \models \varphi$. If $\text{Free}(\varphi) = \emptyset$, then the environment function is irrelevant in the satisfaction relation $(i, \mathcal{E}) \models \varphi$.

Next, we present the STL* fragment in which at most one freeze variable is free. This fragment is already more expressive than STL, and we show later, admits efficient monitoring algorithms. Given a formula φ, the corresponding set of subformulae $\text{Sub}(\varphi)$ is as usual (the formal definition is in the appendix).

Definition 3 (STL_1^* fragment)**.** An STL* formula φ is an STL_1^* formula provided all of the following conditions hold.

(1) For every subformula $\psi \in \text{Sub}(\varphi)$, we have $|\text{Free}(\psi)| \leq 1$, *i.e.*, every subformula can have at most one free variable; and
(2) Corresponding to every subformula $s_k^h.\psi \in \text{Sub}(\varphi)$ involving a freeze operator $s_k^h.$, we have $\text{Free}(\psi)$ to be either $\emptyset$, or $\{s_k^h\}$, that is if $s_k^h.\psi$ is a subformula of φ, then ψ cannot have any free variables apart from s_k^h.
(3) All freeze operators are over unique variables. □

Example 2. $\varphi_1 = s_2^1.(s_2 \geq 2 \rightarrow \Diamond(s_1^1.\Diamond(s_2 \leq s_2^{1*} \wedge s_1 \leq s_1^{1*}+3)))$ is not an STL_1^* formula, as it has the subformula $s_2 \leq s_2^{1*} \wedge s_1 \leq s_1^{1*}+3$ which has two free variables s_1^1, s_2^1. The formula φ_0 from Example 1 is an STL_1^* formula even though it has two freeze variables. □

3 EXPRESSIVENESS OF STL_1^*

In this section we present practical specification examples that can be expressed in STL_1^*. We first present two requirements from [9].

(1) The signal s is oscillating with a period at most 10 time unit

$$\psi_1 = \Box_{[0,T]} \left(\Diamond_{[0,10]} s.\left(\left(\Diamond_{[0,10]} s^* + \delta < s \right) \wedge \left(\Diamond_{[0,10]} s^* > s - \delta \right) \right) \right).$$

(2) s_1 copies the values of s_2 with a delay of $4 \pm \delta$ time units

$$\psi_2 = \Box_{[0,T]} s_2.(\Box_{[4-\delta,4+\delta]} s_2^* = s_1).$$

The first STL_1^* formula says that "for every time instant (from 0 to T), there is a near future (within at most 10) and a value of s, say α, such that a future value of s will be smaller than α within 10 time units, and another future value will be greater than α within 10 time units". The constant δ serves as the threshold for the minimum oscillation amplitude. If we try to express the same requirement using STL, we may consider the following formula:

$$\Box_{[0,T]}((\dot{s} \geq 0 \rightarrow \Diamond_{[0,10]} \dot{s} < 0) \wedge (\dot{s} \leq 0 \rightarrow \Diamond_{[0,10]} \dot{s} > 0)).$$

This formula requires first derivatives, and derivatives are fragile in the presence of noise. Also, this formula lacks the δ factor for specifying the minimum oscillation amplitude.

The second STL_1^* formula shows another example that cannot be expressed using STL. It falls in the category of requirements expressing relations between signal components at different times, but with a time difference that is not a constant. A trick that is often employed in STL is to have a delayed version of the signal as another signal component; but this trick does not work in case the delay is variable as is the case here.

Another important engineering property is checking whether we have a spike within a signal or not. A spike can be defined as a sharp change in a signal value (sudden increase then decrease to the normal value) and in most cases, it is considered undesirable. We can express a spike in STL_1^* by the following formula:

$$\psi_3 = \Diamond_{[0,T]} s.(\Diamond_{[0,w]}(s - s^* > \delta \wedge \Diamond_{[0,w]}|s - s^*| \leq \epsilon)),$$

where $2w$ is the spike width, δ is the spike height, and ϵ is a small value. The formula ψ_3 says that the signal must first rapidly have an increase of at least δ, and after the increase, it must come back down to within a close range ϵ of the pre-spike value. Note that the formula does not assume a fixed position of the spike, it can start anywhere within T time units. It also does not assume anything about the pre-spike value.

In [21], the authors try to express the spike requirement using STL. They give the following formula:

$$\Diamond_{[0,T]}((s_{\text{diff}} > \delta) \wedge \Diamond_{[0,w]}(s_{\text{diff}} < -\delta)),$$

where they define s_{diff} to be the discrete-time derivative of s. The problem with this requirement is that if we have a high sampling rate, it could be the case that we cannot detect a sudden increase in the derivative but we still have a spike.

Another engineering property is to check the settling time. The following STL_1^* checks if a settling time of a signal s is slow:

$$\psi_4 = \Diamond_{[r,T]} s.(|s - s^*| \geq \beta),$$

where r is the latest time the signal needs to settle before it is considered to have a slow settling time and β is a threshold. This requirement cannot be expressed in STL unless we already know

the value that the signal will settle at. A similar formula can be derived to check if the rise time of a signal is acceptable or not.

Here is another set of useful properties that we can express using STL_1^* and that we conjecture cannot be expressed using just STL:

- The signal trend is globally increasing, but locally it could have decreasing segments (e.g., sea level or global temperature): $\psi_5 = \square_{[0,T]} s.\Diamond_{[0,10]}(s^* < s)$.
- At some point in the future, the gap between the current signal value and a future signal value within 10 time units is larger than Δ: $\psi_6 = \Diamond s.\Diamond_{[0,10]}(|s^* - s| \geq \Delta)$.
- All the values of s_2 in the next 10 time units are always within δ from the current value of s_1: $\square_{[0,T]} s_1.(\square_{(0,10]}|s_2 - s_1^*| \leq \delta)$.

STL_1^* is able to specify requirements that cannot be expressed in STL when we compare the signal value to a past frozen value, and the time at which the signal value is frozen needs to be variable.

4 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.

Definition 4 (Syntax Tree)**.** Given an STL* formula φ, the associated abstract syntax tree $\text{AST}(\varphi)$ is defined as follows.

- The nodes of the syntax tree are $\text{Sub}(\varphi)$.
- The root node is φ.
- The edges in the tree are defined by the operator structure:
  - If $s_k^h.\psi \in \text{Sub}(\varphi)$, then $s_k^h.\psi$ has the child ψ.
  - If $\text{op}\,\psi \in \text{Sub}(\varphi)$, for $\text{op} \in \{\neg, \square_I, \Diamond_I\}$, then $\text{op}\,\psi$ has the child ψ.
  - If $\psi_1\,\text{op}\,\psi_2 \in \text{Sub}(\varphi)$, for $\text{op} \in \{\wedge, \vee, \rightarrow, \mathcal{U}_I\}$, then $\psi_1\,\text{op}\,\psi_2$ has the two children ψ_1, ψ_2. □

In order to check whether a timed word satisfies an STL_1^* formula φ, we build on the insight of [11] which noted that we can compute the satisfaction relation for subformulae involving only one free variable (for various word position indices), and after this computation, use these values akin to values computed had the subformula been an STL formula (without any freeze variables). 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 3. Consider the formula from Example 1 ([Running example]), and its associated syntax tree in Figure 1 The formula subscripts correspond to a reverse topological sort of the syntax tree. The freeze variable ordering given by a reverse topological sort is $s^1 <_{\text{revtop}} s^2$. The nodes in $\text{SubTree}_{\varphi_0}(s^1)$ are $\{\varphi_8, \varphi_9\}$. The nodes in $\text{SubTree}_{\varphi_0}(s^2)$ are $\{\varphi_3, \varphi_4, \varphi_5, \varphi_6, \varphi_7\}$. The nodes in $\text{TopSubTree}(\varphi)$ are $\{\varphi_1, \varphi_2\}$. $\text{SubTree}_{\varphi_0}(s^1).\text{root} = \varphi_8$. $\text{SubTree}_{\varphi_0}(s^1).\text{parent} = \varphi_7$. $\text{SubTree}_{\varphi_0}(s^2).\text{root} = \varphi_3$. $\text{SubTree}_{\varphi_0}(s^2).\text{parent} = \varphi_2$. □

Example 4. The syntax tree for non STL_1^* formula φ_1 from Example 2 has 3 subtrees: $\text{SubTree}_{\varphi_1}(s_1) = \{s_1 \leq s_1^* + 3, s_2 \leq s_2^*, s_2 \leq s_2^* \wedge s_1 \leq s_1^* + 3, \Diamond(s_2 \leq s_2^* \wedge s_1 \leq s_1^* + 3)\}$, $\text{SubTree}_{\varphi_1}(s_2) = \{\text{remaining subformulae except } \varphi_1\}$ and $\text{TopSubTree}(\varphi_1) = \{\varphi_1\}$. □

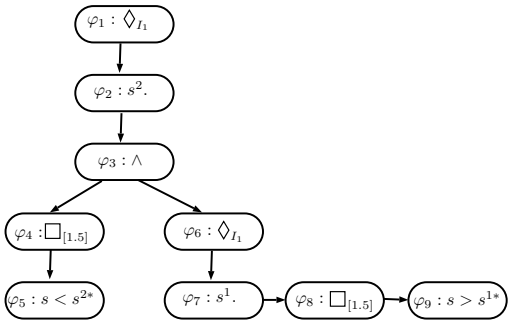


Figure 1: Syntax Tree for Example 1 [Running example].

5 MONITORING ALGORITHMS

Given a trace ρ, and an STL* formula φ, the *monitoring* problem consists of computing the satisfaction value $(0, \mathcal{E}[\equiv \sigma_0]) \models \varphi$, *i.e.*, whether trace ρ starting at position 0 satisfies the given formula for the environment $\mathcal{E}[\equiv \sigma_0]$. In this section, let $i \in [0, |\rho| - 1]$.

A basic STL* monitoring algorithm would be exponential in the number of freeze variables: if we have $|V|$ freeze variables in φ, we 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 $i_1, .., i_{|V|}$ are indices referring to timestamps each ranging from 0 to $|\rho| - 1$, and $j_1, .., j_{|V|}$ are the appropriate signal dimensions. This leads to a $|\rho|^{|V|}$ search space. Since such an exponential time algorithm for general STL* is not practical, we focus on STL_1^* formulae in this paper.

Our algorithms have two core ideas. First, suppose we have an STL_1^* formula φ with just one freeze variable, an intuitive algorithm for the monitoring problem would consist of iterating over all $|\rho|$ instantiations (or environments) of the freeze variable, and, for each environment $\mathcal{E}$, it will have to compute the satisfaction values $(i, \mathcal{E}) \models \psi$ for all trace positions i, for every subformula $\psi \in \text{AST}(\varphi)$. Such algorithm will run in quadratic time in terms of the trace size ($|\rho|$ environments multiplied by $|\rho|$ traces positions). In this work, we will use clever data structures to reuse work across freeze variable instantiations to reduce the $|\rho|^2$ factor.

Second, suppose our STL_1^* formula has more than just one freeze variable. A basic algorithm would still have a dependence on $|\rho|^{|V|}$, like for STL*. However, we show that we can avoid the exponential complexity by leveraging independence of the freeze variables. The *isolation idea*, as in [11], is to work on $\text{SubTree}_\varphi(x_k)$ in isolation, provided the satisfaction relation for all subformulae $x_{k'}.\psi_{x_{k'}}$ have been computed for all freeze variables $x_{k'}$ such that $x_{k'} <_{\text{revtop}} x_k$, where $<_{\text{revtop}}$ denotes a reverse topological ordering of the freeze variables in the syntax tree for φ (in other words, we freeze a single freeze variable at a time to avoid the exponential complexity). This can not be done in the case of a general STL* formula. Conceptually, when working on the $\text{SubTree}_\varphi(x_k)$ in isolation, the idea is to compute, for all i, the satisfaction values for $(i, \mathcal{E}[x_k = \sigma_i(j_k)]) \models \psi$ for every subformula $\psi \in \text{SubTree}_\varphi(x_k)$, for every possible freeze variable binding $x_k = \sigma_i(j_k)$. Note here that $\text{Free}(\psi)$ can only be $\emptyset$ or $\{x_k\}$ since φ is an STL_1^* formula; and hence the environment needed is only $\mathcal{E}[x_k = \sigma_i(j_k)]$; compared to environments looking like $\mathcal{E}[x_1 = \sigma_{i_1}(j_1), x_2 = \sigma_{i_2}(j_2) \ldots x_{|V|} = \sigma_{i_{|V|}}(j_{|V|})]$ for general STL* formulae. The environment $\mathcal{E}[x_k = \sigma_i(j_k)]$ only assigns

the value $\sigma_i(j_k)$ to the frozen value x_k^*, for the remaining freeze variables $x \neq x_k$, we do not need to assign any frozen value since the values assigned to these freeze variables do not influence the value of the satisfaction relation of any subformula in $\text{SubTree}_\varphi(x_k)$.

In the next two subsections, we present two algorithms that both use the *isolation* idea that we mentioned above. The second optimized algorithm additionally uses the first idea mentioned in the beginning of the section: for a given freeze variable in a given subtree at a given instantiation, it uses (a) the information collected from the previous instantiation to calculate the information needed for the current instantiation more efficiently (the first algorithm calculates the information of each instantiation from scratch), and (b) more sophisticated data structures.

5.1 STL_1^* Polynomial Time Algorithm

This first algorithm is inspired by the work in [11]. The algorithm starts by calculating the values $(i', \mathcal{E}[x_1 := \sigma_i(j_1)]) \models \varphi_{k'}$ for every subformula $\varphi_{k'} \in \text{SubTree}_\varphi(x_1)$, for every environment $\mathcal{E}[x_1 := \sigma_i(j_1)], \forall i$ and every $i' \in [i, |\rho| - 1]$ (here $\sigma(j_1)$ is the signal dimension corresponding to x_1). Then it calculates the values $(i'', \mathcal{E}[x_2 := \sigma_i(j_2)]) \models \varphi_{k''}$ for every subformula $\varphi_{k''} \in \text{SubTree}_\varphi(x_2)$, for every environment $\mathcal{E}[x_2 := \sigma_i(j_2)], \forall i$ and every $i'' \in [i, |\rho| - 1]$ (similarly, $\sigma(j_2)$ is the signal dimension corresponding to x_2) and so on till it reaches $\text{TopSubTree}(\varphi)$. The indices 1,2 in x_1, x_2 do not refer to the first or second dimension of the signal, they refers to order of the freeze variables in $\text{AST}(\varphi)$: $x_1 <_{\text{revtop}} x_2$.

However, this algorithm, for a given instantiation, is not always able to calculate the satisfaction relation of a subformula for i timestamps in $O(i)$ time; in other words, computing the satisfaction relation of a given subformula for a given environment at a given timestamp takes more than $O(1)$. To better explain this, let us recall the recursive definition of the until operator: $(i, \mathcal{E}) \models \varphi_1 \mathcal{U}_I \varphi_2$ iff

- $0 \in I$ and $(i, \mathcal{E}) \models \varphi_2$,
- or $i < |\rho| - 1$ and $(i, \mathcal{E}) \models \varphi_1$ and $(i+1, \mathcal{E}) \models \varphi_1 \mathcal{U}_{I'} \varphi_2$ where $I' = I - \tau_{i+1} + \tau_i$.

This definition implies that, if we want to compute, for example, the satisfaction relation $(i, \mathcal{E}) \models \varphi_2 = \varphi_0 \mathcal{U}_{[3,5]} \varphi_1, \forall i \in [0, |\rho| - 1]$ for a given environment $\mathcal{E}$, with a uniformly sampled trace with a sampling rate of 1 second, we will eventually be calculating the following satisfaction relations $(i, \mathcal{E}) \models \varphi_0 \mathcal{U}_{[2,4]} \varphi_1$, $(i, \mathcal{E}) \models \varphi_0 \mathcal{U}_{[1,3]} \varphi_1$ and $(i, \mathcal{E}) \models \varphi_0 \mathcal{U}_{[0,2]} \varphi_1$ for some i values. The subformulae are not in the syntax tree and we need to calculate these satisfaction relations as intermediate steps.

As a second example for a non-uniformly sampled trace, suppose we have the formula $\varphi_2 = \varphi_0 \mathcal{U}_{[7,9]} \varphi_1$ and a trace of 5 timestamps 0, 2, 5, 7 and 10. If we want to calculate $(2, \mathcal{E}) \models \varphi_2$, if $(2, \mathcal{E}) \models \varphi_0$ is TRUE, we will have to check $(3, \mathcal{E}) \models \varphi_0 \mathcal{U}_{[5,7]} \varphi_1$. However, we do not have that value. To get it, if $(3, \mathcal{E}) \models \varphi_0$ is TRUE, we will need to also check $(4, \mathcal{E}) \models \varphi_0 \mathcal{U}_{[2,4]} \varphi_1$. Here, we reached the trace end (timestamp 10, index 4) and we can get the value of $(4, \mathcal{E}) \models \varphi_0 \mathcal{U}_{[2,4]} \varphi_1$ without additional steps.

However, in practice, the trace is much longer and to compute $(i, \mathcal{E}) \models \varphi_0 \mathcal{U}_I \varphi_1$, for most i values, we would reach an interval for the until operator that starts with 0 before we reach the end of the trace. The steps that we just showed in this second example must be done for every time stamp in the trace in order to get $(i, \mathcal{E}) \models \varphi_0 \mathcal{U}_I \varphi_1$ for every i. This brings the complexity of this polynomial monitoring algorithm to

$$O\left(c \cdot |\,\text{AST}(\varphi)| \cdot |V| \cdot |\rho|^2\right), \tag{1}$$

where $|\,\text{AST}(\varphi)|$ is the size of $\text{AST}(\varphi)$ and $c = \lceil a/\Delta \rceil$ where a is the largest constant occurring in the temporal operators in φ, and Δ is the smallest difference between two consecutive timestamps.

Algorithms 1 and 2 correspond to the algorithm that we just described above.

Algorithm 1: ComputeSTL*

Input: φ_j, i, t
Output: TRUE or FALSE
if $\varphi_j = s_k^* \sim s_{k'} \pm r$ **then return** $\sigma_t(k) \sim \sigma_i(k') \pm r$
if $\varphi_j = \neg\varphi_m$ **then return** $\neg M[m, i]$
if $\varphi_j = \varphi_m \vee \varphi_n$ **then return** $M[m, i] \vee M[n, i]$
if $\varphi_j = \varphi_m \mathcal{U}_I \varphi_n$ **then**
  if $0 \in I$ *and* $M[n, i] = T$ **then return** TRUE
  else if $i < |\rho| - 1$ *and* $M[m, i] = T$ *and* ComputeSTL*$(\varphi_m \mathcal{U}_{I - \tau_{i+1} + \tau_i} \varphi_n, i + 1, t) = T$ **then return** TRUE
  else return FALSE

Algorithm 2: STL_1^* Polynomial Time Monitoring

Input: $\text{AST}(\varphi)$, ρ
Output: intvl (φ_1)
for $k \leftarrow 1$ *to* $|V|$ **do**
  for $t \leftarrow 0$ *to* $|\rho| - 1$ **do**
    for $j \leftarrow \text{SubTree}_\varphi(x_k)$. max *down to* $\text{SubTree}_\varphi(x_k)$. min **do**
      for $i \leftarrow |\rho| - 1$ *down to* t **do**
        $M[j, i] \leftarrow$ ComputeSTL*(φ_j, i, t)

if $|\,\text{TopSubTree}(\varphi)| \neq 1$ **then**
  for *each subformula* $\varphi_j \in \text{TopSubTree}(\varphi)$ **do**
    for $i \leftarrow |\rho| - 1$ *down to* t **do**
      $M[j, i] \leftarrow$ ComputeSTL*$(\varphi_j, i, 0)$ // The 3rd argument for ComputeSTL* does not matter here since we do not have any signal constraint in the TopSubTree.
return $M[1, 0]$

5.2 Optimized STL_1^* Algorithm

We now present an opimized algorithm in which we improve the theoretical performance of the algorithm in Subsection 5.1 further — we remove the c factor in Equation 1, and heuristically lower the $|\rho|^2$ factor. First, we define notations and the data structures that we are going to use.

5.2.1 Notation. For any STL* subformula ψ, we note

- $\text{point}(\psi)$: a list of length $|\rho|$ of TRUE and FALSE values where for each $0 \leq i \leq |\rho| - 1$, each value represents whether $(i, \mathcal{E}) \models \psi$ or not for a given environment $\mathcal{E}$.
- $\text{intvl}(\psi)$: 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 $a_1 \leq a_2 < b_1 \leq b_2 \cdots < z_1 \leq z_2$, covering any sequence of TRUE's appearing in $\text{point}(\psi)$.

Example 5. Suppose we have the following trace $(\sigma_i, \tau_i), i \in [0, 10]$: (5,0), (3,1), (7,2), (-2,6), (-5,4), (3,5), (-1,6), (3,7), (4,8), (5,9), (6,10) and the subformula $\varphi_0 = s \geq 0$, then:

- $\text{point}(\varphi_0) = [T, T, T, F, F, T, F, T, T, T, T]$.
- $\text{intvl}(\varphi_0) = [0, 2], [5, 5], [7, 10]$. □

5.2.2 Data Structures.

- Timestamps array: an array of size $|\rho|$ which have the timestamps values.
- Signal dimension array: For each signal dimension, we use an array of size $|\rho|$ to store the signal dimension values.
- Doubly Linked List sorted($\sigma(k)$): For a signal dimension $\sigma(k)$, we use sorted($\sigma(k)$) to store the sample values in increasing sorted order, but we also use a reference to link each node in this linked list to its corresponding node or *position index* in the signal dimension array (in other words, we keep track of the original position before sorting of a given signal dimension value, check Figure 2). With that, we guarantee $O(1)$ access time in the list. Each node in the signal dimension array has a reference to the node with the same value in the sorted list and vice versa. In case of a value appearing multiple times in a signal dimension, that value will appear the same number of times in the sorted list and we assign each one to its corresponding position index.
- $\text{point}(\varphi_j)$: an Array of size $|\rho|$.
- $\text{intvl}(\varphi_j) = [\text{start}_j(0), \text{end}_j(0)], [\text{start}_j(1), \text{end}_j(1)], \ldots, [\text{start}_j(n), \text{end}_j(n)]$.
- start_j and end_j: arrays (of size $n \leq |\rho|$ each). We use two basic operations on start_j and end_j, *add* and *remove*. For *remove*, it is a "lazy" remove: instead of removing an element right away when we call *remove*, we only do that once we call sort on the array.
- flip_j is an array of size $|\rho|$ corresponding to each signal constraint φ_j.

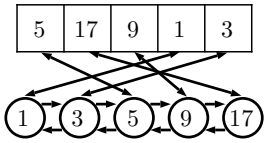


Figure 2: A signal dimension Array and the corresp. sorted list

5.2.3 Algorithm Overview. When trying to calculate $(i, \mathcal{E}) \models \psi$ for any given φ and any environment $\mathcal{E}$, instead of iterating over all the timestamps $\forall i$ to calculate the different satisfaction relations, we iterate over the intervals in $\text{intvl}(\psi)$ where in practice the size of $\text{intvl}(\psi)$ (the size of $\text{intvl}(\psi)$ is the number of intervals in $\text{intvl}(\psi)$) is way smaller than the number of timestamps. This will give us the same results in a reduced number of computations. In fact, both algorithms will go over the same number of instantiations for each freeze variable, the only difference is that our optimized algorithm takes $O(|\text{intvl}(\psi)|)$ when calculating the satisfaction relation of a given subformula for a given environment while the polynomial one takes $O(c.i)$). Let us consider example 5, and suppose we want to calculate $(i, \mathcal{E}) \models \square_{[1,2]}\varphi_0, \forall i$, the polynomial STL_1^* algorithm will need to calculate 10 satisfaction relations (one for each i) while the new algorithm will calculate just 3 (one for each interval in $\text{intvl}(\varphi_0)$, it has just 3 intervals). Also, in some cases, when a subformula ψ is either a signal constraint or of the form $x_k.\psi'$, we need to calculate $\text{point}(\psi)$ (the vector $\text{point}(\psi)$ represents $(i, \mathcal{E}) \models \psi, i \geq i'$ for a given $i' \in [0, |\rho|-1]$ and a given $\mathcal{E}$) and not just $\text{intvl}(\psi)$. The nature of the trace ρ (pointwise semantics and discrete timestamps and not a continuous signal) is the main reason why we have to go over $\text{point}(\psi)$ as a first step and not directly calculate $\text{intvl}(\psi)$, in other words, we cannot calculate $\text{intvl}(\psi)$ without calculating $\text{point}(\psi)$ first, for ψ of these forms. For the case of a signal constraint, we try to update a limited number of values in $\text{point}(\psi)$ and not iterate over all values of i.

1. For each freeze variable $x_k, 1 \leq k \leq |V|$ such that $x_{|V|} > \ldots > x_2 > x_1$
   - 1.1. For instantiation 0 (initialization)
     - Compute $point(\psi)$ for each signal constraint $\varphi_j \in \text{SubTree}_\varphi(x_k)$.
     - Compute $intvl(\psi)$ for each $\psi \in \text{SubTree}_\varphi(x_k)$.
     - Assign a TRUE or FALSE value to $point(\text{SubTree}_\varphi(x_k).parent)[0]$ depending on $intvl(SubTree_\varphi(x_k).root)$ starts with 0 or not.
   - 1.2. For each instantiation $1 \leq i \leq |\rho| - 1$
     - Update values $point(\psi)[l]$, $i \leq l \leq |\rho| - 1$ (some values, not all), for each signal constraint $\psi \in \text{SubTree}_\varphi(x_k)$.
     - Update $intvl(\psi)$ for each $\psi \in \text{SubTree}_\varphi(x_k)$.
     - Assign a TRUE or FALSE value to $point(\text{SubTree}_\varphi(x_k).parent)[i]$ depending on $intvl(SubTree_\varphi(x_k).root)$ starts with τ_i or not.
2. Compute $intvl(\psi)$ for each $\psi \in \text{TopSubTree}(\varphi)$

Figure 3: Optimized algorithm overview

Algorithm Logic: Figure 3 gives an overview of our algorithm steps. If we consider our running example 1, the algorithm calculates $\text{point}(\varphi_9)$, $\text{intvl}(\varphi_9)$ then $\text{intvl}(\varphi_8)$ for each instantiation of s_1 to $\sigma_i(1)$, for all i and with each instantiation, appends a new value to $\text{point}(\varphi_7)$ based on the condition "$\text{intvl}(\varphi_8)$ starts with τ_i or not". Then, our algorithm does the same steps for all the instantiations $\sigma_i(2)$ of the second freeze variable s_2. Finally, it finishes by calculating $\text{intvl}(\varphi_1)$.

5.2.4 `Transform` *Algorithm.* Given an input $\text{point}(\varphi_j)$ and an integer i, this simple algorithm transforms the values in $\text{point}(\varphi_j)$ starting from position i to $\text{intvl}(\varphi_j)$.

Example 6. Suppose we have a uniformly sampled trace with a sampling rate equal to 1 second and $i = 5$. Then our algorithm transforms $\text{point}(\varphi_1) = [F, F, F, T, T, T, T, T, F, F, T, T, T, T]$ to $\text{intvl}(\varphi_1) = [5, 7], [10, 13]$. □

Algorithm 3: `Transform`

Input: $\text{point}(\varphi_j), i$
Output: $\text{intvl}(\varphi_j)$

```
if point(φ_j)[i] = TRUE then start_j . add(τ_i)
for l ← i + 1 to |ρ| − 1 do
    if point(φ_j)[l] = TRUE and point(φ_j)[l − 1] = FALSE then
        start_j . add(τ_l)
    if point(φ_j)[l] = FALSE and point(φ_j)[l − 1] = TRUE then
        end_j . add(τ_{l−1})
if point(φ_j)[|ρ| − 1] = TRUE then end_j . add(τ_{|ρ|−1})
return intvl(φ_j)
```

Algorithm 4: STL_1^* Optimized Monitoring

Input: AST(φ), ρ
Output: intvl (φ_1)

1 intvl (φ') ← `Transform`(point(φ'), 0), $\forall\varphi'$ signal pred.
2 **for** *each signal dimension* $\sigma(i)$ **do**
3 sorted($\sigma(i)$) ← sorted $\sigma(i)$ values
4 **for** $k \leftarrow 1$ *to* $|V|$ **do**
5 **for** $j \leftarrow$ SubTree$_\varphi(x_k)$.max *down to* SubTree$_\varphi(x_k)$.min **do**
6 **if** φ_j *is a signal constraint on the freeze variable* x_k **then**
7 Calculate flip$_j$[0], point(φ_j)
8 intvl (φ_j) ← `Transform`(point(φ_j), 0)
9 **else** intvl (φ_j) ← `ComputeIntervals`(φ_j, 0)
10 **if** intvl (SubTree$_\varphi(x_k)$.root) *starts with* 0 **then** point(SubTree$_\varphi(x_k)$.parent)[0] ← TRUE
11 **else** point(SubTree$_\varphi(x_k)$.parent)[0] ← FALSE
12 **for** $i \leftarrow 1$ *to* $|\rho| - 1$ **do**
13 **for** $j \leftarrow$ SubTree$_\varphi(x_k)$.max *down to* SubTree$_\varphi(x_k)$.min **do**
14 **if** φ_j *is a signal constraint* **then** intvl (φ_j) ← `UpdateSignalConstraint`(φ_j, i)
15 **else** intvl (φ_j) ← `ComputeIntervals`(φ_j, i)
16 **if** intvl (SubTree$_\varphi(x_k)$.root) *starts with* τ_i **then** point(SubTree$_\varphi(x_k)$.parent)[i] ← TRUE
17 **else** point(SubTree$_\varphi(x_k)$.parent)[i] ← FALSE
18 intvl (SubTree$_\varphi(x_k)$.parent) ← `Transform`(point(SubTree$_\varphi(x_k)$.parent), 0)
19 **if** |TopSubTree(φ)| ≠ 1 **then**
20 **for** *each subformula* $\varphi_j \in$ TopSubTree(φ) **do**
21 intvl (φ_j) ← `ComputeIntervals`(φ_j, 0)
22 **return** intvl (φ_1)

5.2.5 STL_1^* Monitoring Algorithm.

This is the main algorithm and it works as follows: First, It calculates point(φ') for any φ' signal predicate (Line 1) and sorted($\sigma(j)$) for every signal dimension $\sigma(j)$ (Line 2). Then, for each freeze variable x_k (Line 4) (here, we use the freeze variable ordering $x_1 < \cdots < x_{|V|}$):

- The algorithm calculates the values of each subformula in the SubTree$_\varphi(x_k)$ for each timestamp τ_i, $i \in [0, |\rho| - 1]$ corresponding to the instantiation of x_k to $\sigma_0(j_k)$ (Lines 5-9), calculates the value of flip$_j$[0] for every subformula φ_j signal constraint and assigns a TRUE or FALSE value to point(SubTree$_\varphi(x_k)$.parent)[0], Lines 10-11 (point(SubTree$_\varphi(x_k)$.parent)[0] represents the first value in the vector point(SubTree$_\varphi(x_k)$.parent) corresponding to the parent node of SubTree$_\varphi(x_k)$ as defined in section 4).
- Then, for each timestamp τ_i, $i \in [1, |\rho| - 1]$ (Line 12), the algorithm calculates the new values of each subformula for each instantiation i (Lines 13-15) and assigns a value to point(SubTree$_\varphi(x_k)$.parent)[i] (Lines 16-17).
- Finally, once we have all the values for point(SubTree$_\varphi(x_k)$), the algorithm calculates intvl (SubTree$_\varphi(x_k)$) (Line 18).

Once the algorithm finishes the for loop in Line 4 (finishes with all SubTree$_\varphi(x_k)$'s for each freeze variable x_k), the final step would be to calculate the values of the TopSubTree(φ).

5.2.6 `UpdateSignalConstraint` *Algorithm.*

Let φ_j be a signal constraint of the form $s_k^* \sim s_{k'} \pm r$, the main goal of this algorithm is to update the values of point(φ_j) and intvl (φ_j) for the instantiation $i+1$ given the values of point(φ_j) and intvl (φ_j) for the instantiation i. In other words, we want to calculate the satisfaction relation $(i', \mathcal{E}[s_k := \sigma_{i+1}(k)]) \models \varphi_j$ for any $i' \geq i+1$ given the satisfaction relation $(i', \mathcal{E}[s_k := \sigma_i(k)]) \models \varphi_j, i' \geq i$.

flip$_j$[i] is the position index where the signal constraint φ_j (in sorted($\sigma(k)$)) switches values from TRUE to FALSE or the opposite in the i^{th} instantiation. Here, if we interpret the signal constraint φ_j as a function of $s_{k'}$, and since the signal dimension values are sorted in sorted($\sigma(k)$), we can see that flip$_j$[i] represents a threshold for when we reach a value in sorted($\sigma(k)$) for which φ_j is TRUE (resp. FALSE) for all the remaining values in sorted($\sigma(k)$) and FALSE (resp. TRUE) for all the previous values.

The algorithm uses flip$_j$'s to track the values that changed from an instantiation to the next one. We will consider the example from 2 for a better explanation and let us suppose we have the following signal constraint $\varphi_5 = s \geq s^* + 2$ where $s^* = 5$ (instantiation 0). Then we have flip$_5$[0] = 3. For the next instantiation, we have $s^* = 17$ and flip$_5$[1] = 5. The position indices between flip$_5$[0] and flip$_5$[1] − 1 are 2 (for the signal value 9) and 1 (for the signal value 17). Now back to the algorithm, it first removes the no longer needed value $\sigma_i(k)$ from sorted($\sigma(k)$) (that value is not needed when calculating the satisfaction relation $(i', \mathcal{E}) \models s_k^* \sim s_{k'} \pm r$ for $i' \geq i+1$) and updates flip$_j$[i] accordingly (Lines 1-2). Then, it calculates the new value flip$_j$[$i+1$]. Given flip$_j$[i] and flip$_j$[$i+1$], the algorithm updates certain values in point(φ_j), start$_j$ and end$_j$ (values corresponding to position indices between flip$_j$[i] and flip$_j$[$i+1$] − 1 in sorted($\sigma(k)$)) by calling `Sub-Update` 6.

Finally, the algorithm either sorts the values in start$_j$ and end$_j$ to get intvl (φ_j) (Lines 6-8) (since the values in intvl (φ_j) are initially from the previous instantiation, it could be the case that the first interval in intvl (φ_j) starts with τ_{i-1}, we use the operation in line 8 to make sure that it starts with $\tau_{i'}$ where $i' \geq i$) or just calculates start$_j$ and end$_j$ from scratch using point(φ_j) (Lines 9-11), depending on which operation is estimated to be faster. In fact, in some cases, start$_j$ and end$_j$ can be too long (we use the condition in line 6) and it will be better to remove all the values from start$_j$ and end$_j$, and iterate over point(φ_j) to get the new values sorted (Line 7 takes $O(size(\text{start}_j).\log(size(\text{start}_j)))$ while Line 11 takes $O(|\rho|)$).

5.2.7 `Sub-Update` *Algorithm.*

Given point(φ_j), start$_j$, end$_j$ and a position index l, the goal of this algorithm is to, first, update the value point(φ_j)[l] corresponding to the satisfaction relation $(l, \mathcal{E}[s_k := \sigma_i(k)]) \models \varphi_j$ (i is the current value when `UpdateSignalConstraint` calls `Sub-Update`). And, second, make the necessary changes to start$_j$ and end$_j$ so that intvl (φ_j) is also updated and keeping track of the changes happening to point(φ_j).

For each position index l' that comes between flip$_j$[$i-1$] and l in sorted($\sigma(k)$), `Sub-Update` has been already called multiple times by the *for* loop in Line 5 of algorithm 5 and we already have updated all the values of point(φ_j)[l'] for each l' from TRUE to FALSE or the opposite (either all the values become TRUE or the opposite) corresponding to the satisfaction relation $(l', \mathcal{E}[s_k := \sigma_i(k)]) \models \varphi_j$, similarly, updates have been made to intvl (φ_j) with each call (check the last paragraph of this section for an example).

Algorithm 5: `UpdateSignalConstraint`

Input: intvl (φ_j) in $i^{\text{th}} - 1$ instantiation, i
Output: intvl (φ_j) in i^{th} instantiation
1 **if** $\text{flip}_j[i-1] \geq \tau_{i-1}$ **then** $\text{flip}_j[i-1] \leftarrow \text{flip}_j[i-1] - 1$
2 $\text{sorted}(\sigma(k))$. *remove*$(\sigma_{i-1}(k))$
3 Calculate $\text{flip}_j[i]$
4 **for** *each position index* l *between* $\text{flip}_j[i-1]$ *and* $\text{flip}_j[i] - 1$ *in* $\text{sorted}(\sigma(k))$ **do**
5 $\quad \text{start}_j, \text{end}_j, \text{point}(\varphi_j) \leftarrow$ `Sub-Update`$(\tau_l, \text{start}_j, \text{end}_j)$
6 **if** *size*(start_j) .*log*(*size*$(\text{start}_j)) < |\rho|$ **then**
7 $\quad$ sort start_j and end_j
8 $\quad \text{intvl}(\varphi_j) \leftarrow \text{intvl}(\varphi_j) \cap [\tau_i, \tau_{|\rho-1|}]$
9 **else**
10 $\quad$ empty start_j and end_j
11 $\quad \text{intvl}(\varphi_j) \leftarrow$ `Transform`$(\text{point}(\varphi_j), i)$
12 **return** $\text{intvl}(\varphi_j)$

The algorithm updates the value $\text{point}(\varphi_j)[l]$ from TRUE to FALSE (or the opposite) (Lines 6 and 12), and also it checks the values $\text{point}(\varphi_j)[l+1]$ and $\text{point}(\varphi_j)[l-1]$ to update start_j and end_j going over multiple possible cases (Lines 1-5 and 7-11).

Let us consider the example where $\text{intvl}(\varphi_1) = [2, 10], [20, 35]$ (in other words $\text{start}_1 = [2, 20]$ and $\text{end}_1[10, 35]$) and suppose the value $\text{point}(\varphi_1)[8]$ changes from TRUE to FALSE. Then the algorithm will change the value $\text{point}(\varphi_1)[8]$ to FALSE (Line 6). The conditions in Lines 2 and 4 are satisfied so the algorithm will add the value 9 to start_1 and the value 7 to end_1 end we end up with $\text{start}_1 = [2, 20, 9]$ and $\text{end}_1[10, 35, 7]$. and once we sort end_1 and start_1 (this is done in the `UpdateSignalConstraint` algorithm), we get $\text{intvl}(\varphi_1) = [2, 7], [9, 10], [20, 35]$.

Note here, with each call of `Sub-Update`, the algorithm makes simple changes to start_j and end_j while making sure to keep both arrays (start_j and end_j) of same size. The algorithm either adds one value to both arrays, removes one value from both arrays, or, adds one value to one of the arrays and removes another value from the same array. Having the same size for start_j and end_j guarantees that we have the correct values for $\text{intvl}(\varphi_j)$ when we sort start_j and end_j in Line 7 of `UpdateSignalConstraint`.

Algorithm 6: `Sub-Update`

Input: $\tau_l, \text{start}_j, \text{end}_j$
Output: $\text{start}_j, \text{end}_j, \text{point}(\varphi_j)$
1 **if** $\text{point}(\varphi_j)[l] =$ TRUE **then**
2 $\quad$ **if** $\text{point}(\varphi_j)[l+1] =$ TRUE **then** start_j . *add*(τ_{l+1})
3 $\quad$ **else** end_j . *remove*(τ_l)
4 $\quad$ **if** $\text{point}(\varphi_j)[l-1] =$ TRUE **then** end_j . *add*(τ_{l-1})
5 $\quad$ **else** start_j . *remove*(τ_l)
6 $\quad \text{point}(\varphi_j)[l] \leftarrow$ FALSE
7 **if** $\text{point}(\varphi_j)[l] =$ FALSE **then**
8 $\quad$ **if** $\text{point}(\varphi_j)[l+1] =$ FALSE **then** end_j . *add*(τ_l)
9 $\quad$ **else** start_j . *remove*(τ_{l+1})
10 $\quad$ **if** $\text{point}(\varphi_j)[l-1] =$ FALSE **then** start_j . *add*(τ_l)
11 $\quad$ **else** end_j . *remove*(τ_{l-1})
12 $\quad \text{point}(\varphi_j)[l] \leftarrow$ TRUE
13 **return** $\text{start}_j, \text{end}_j, \text{point}(\varphi_j)$

5.2.8 `ComputeIntervals` *Algorithm.* In this section, we show how we compute, for a given environment $\mathcal{E}$, $\text{intvl}(\varphi_j)$ of subformula φ_j with boolean or temporal operators. The idea is based on [24], we slightly modify it to make it work for pointwise semantics. Suppose we have two traces with different sampling rates. The first one, ρ_1, is uniformly sampled of length 100 and the sampling rate is 1 second. And the second one, ρ_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 ρ_1 and ρ_2.

Boolean operators. For Boolean operators, the computation is straightforward. We have the following:
∘ For the uniformly sampled trace ρ_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 ρ_2:

- $\text{intvl}(\neg\varphi_1) = [0, 1], [11, 17], [40, 40]$
- $\text{intvl}(\varphi_1 \vee \varphi_2) = [2, 15], [20, 35]$
- $\text{intvl}(\varphi_1 \wedge \varphi_2) = [7, 10]$

Computing $\text{intvl}(\varphi_j)$ for Boolean operators takes $O(|\text{intvl}(\varphi_j)|)$.

Temporal operators. To treat temporal operators, we need to define the following $[a, b]$-*back shifting* operation as in [24]:

Definition 5. 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 $[\tau_i, \tau_j], k \leq i, j \leq |\rho - 1|$ included in I. □

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 $O(|\text{intvl}(\varphi)|.\log(|\rho|))$ where the $O(\log(|\rho|))$ is paid to find the largest possible interval $[\tau_i, \tau_j], k \leq i, j \leq |\rho - 1|$ included in I for each interval I in $\text{intvl}(\varphi)$ using binary search. Or, we can simply iterate over all the timestamps in ρ 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 $O(|\rho|)$.

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 the uniformly sampled trace ρ_1, $\text{intvl}(\Diamond_{[1,3]}\varphi_1) = [0, 9], [17, 34]$. This will take $O(|\text{intvl}(\varphi)|)$.
- For the non-uniformly sampled trace ρ_2, $\text{intvl}(\Diamond_{[1,3]}\varphi_1) = [0, 8], [17, 30]$. This will take $O(|\rho|)$ or $O(|\text{intvl}(\varphi)|.\log(|\rho|))$ (depending on which method used).

Always operator $\Box_{[a,b]}$*:* For $\Box_{[a,b]}\varphi$, we can abuse notation and define it as follows $\text{intvl}(\Box_{[a,b]}\varphi) = \text{intvl}(\varphi) \ominus [b, a]$. Note in case of $\text{intvl}(\varphi) = [m, n]$ and $n - b < m - a$, $\text{intvl}(\Box_{[a,b]}\varphi) = \emptyset$.

- For the uniformly sampled trace ρ_1, intvl $(\square_{[1,6]}\varphi_1) = [1,4]$, $[19,29]$.
- For the non-uniformly sampled trace ρ_2, intvl $(\square_{[1,6]}\varphi_1) = [1,4]$, $[20,27]$.

Until operator $\mathcal{U}_{[a,b]}$: For the *until* operator $\varphi_1 \mathcal{U}_{[a,b]} \varphi_2$, we will use the same claim used in [24].

Claim. 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_1 and J in φ_2, 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 ρ_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) $[20,35] \cap [7,15] = \emptyset$
$\Rightarrow$ intvl $(\varphi_1 \mathcal{U}_{[2,4]} \varphi_2) = [3,8]$.
And, for the non-uniformly sampled trace ρ_2: we have 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 $O(|\rho|)$ or $O(|\text{intvl}(\varphi_1)| + |\text{intvl}(\varphi_2)|.\log(|\rho|))$

Algorithm 7: `ComputeIntervals`

Input: φ_j, i
Output: intvl (φ_j)
if $\varphi_j = \neg\varphi_m$ // intvl $(\varphi_m) = ([\tau_{m_a}, \tau_{m_b}] \ldots [\tau_{m_y}, \tau_{m_z}])$ **then**
  intvl $(\varphi_j) \leftarrow ([\tau_i, \tau_{m_a - 1}], [\tau_{m_b+1}, \tau_{m_c-1}] \ldots [\tau_{m_z+1}, \tau_{|\rho|-1}])$
if $\varphi_j = \varphi_m \vee \varphi_n$ **then**
  intvl $(\varphi_j) \leftarrow (\text{intvl}(\varphi_m) \cup \text{intvl}(\varphi_n)) \cap [\tau_i, \tau_{|\rho|-1}]$
if $\varphi_j = \varphi_m \mathcal{U}_{[a,b]} \varphi_n$ **then**
  intvl $(\varphi_j) \leftarrow ()$
  for *each interval I in* intvl (φ_m) **do**
    for *each interval J in* intvl (φ_n) **do**
      intvl $(\varphi_j).append\Big(\big((I \cap J) \ominus [a,b]\big) \cap I\Big)$
  intvl $(\varphi_j) \leftarrow \text{trim}^i(\text{intvl}(\varphi_j))$
return intvl (φ_j)

Pushing further more. For the trim operator, we can still improve the complexity. In fact, the goal of the trim operation is, given an ordered trace and a set of ordered intervals, to try and match the bounds of each interval to the nearest timestamps from the trace (timestamps must be included in the interval). We could simply walk through the trace as we described it in **Note 2**. Or, even better, we could use an exponential search [7].

Suppose we have the following trace: 0, 1, 3, 4, 6, 9, 13, 15, 16, 20, 22, 25, 28, 30, 35, and we have the following two intervals we are trying to trim: [12,17] and [21,30]. First, we start with the right bound of the first interval, 12. We compare 12 with the first timestamp in the trace, 0, 12 is bigger so we move to the next timestamp. 12 is bigger than 1 so we jump two timestamps. 12 is bigger than 4 so we jump 4 timestamps (we exponentially increase the jump size each time). 15 is bigger than 12 so we stop and we search for our target timestamp (which is 13 in this case) using binary search in the values ranging between 4 and 15.

Once we find the target timestamp, the algorithm will move on to the second bound which is 17, and repeat what we did in the first step. The only difference now is that we start from the timestamp 15 this time and not 0. Similarly, the same steps will be done for all the bounds of the different intervals that we have.

Clearly, the advantage of this algorithm is that it can skip some timestamps and it does not need to iterate over all the timestamps. This algorithm can lower all the $O(|\rho|)$ (or $O(|\text{intvl}(\varphi)|.\log|\rho|)$) complexities that we had in this section to $O(\log(i_1) + \log(i_2 - i_1) + \cdots + \log(i_{2.|\text{intvl}(\varphi)|} - i_{2.|\text{intvl}(\varphi)|-1}))$ where the i's are the indices of the different target timestamps. We can also simply write down the previous complexity as $O(\min(|\rho|, |\text{intvl}(\varphi)|.\log|\rho|))$. In other words, the exponential search algorithm guarantees us to have the best out of the two complexities $O(|\rho|)$ and $O(|\text{intvl}(\varphi)|.\log|\rho|)$.

6 EXAMPLE

In this section, we will go over the running steps of our algorithm for the following formula:

$$\varphi_0 = \Diamond_{I_2} s^2.(\square_{[1,5]} s < s^{2*} \wedge \Diamond_{I_1} s^1.(\square_{[1,5]} s > s^{1*}))$$

We will consider a uniformly sampled trace with a sampling rate of 1 second. Lines are shown in order as in how the algorithm runs. If we want to consider a different example with a non-uniformly sampled trace, the steps would be exactly the same with just one small difference: for any subformula ψ, calculating intvl (ψ) will be slightly different (the difference is when to apply the trim operator, this was explained in details in section 5.2.8).

In this example, our signal has just one component and it has the following values: $s = (2,5,7,10,15,13,11,6,3,1,7)$. The algorithm's first step is to sort the signal: sorted$(s) = (1,2,3,5,6,7,7,10,11,13,15)$.

For the first freeze variable s^1, we run the different instantiations $i, i \in [0, |\rho|-1]$. Below is a breakdown of each variable the algorithm calculates for the different instantiations:

Instantiation 0: $s^{*1} = 2$:
$\text{flip}_9[0] = 2$, point$(\varphi_9) = [F,T,T,T,T,T,T,T,T,F,T]$
intvl $(\varphi_9) = [1,8]$, $[10,10]$, intvl $(\varphi_8) = [0,3]$
Does intvl (φ_8) start with 0? Yes $\Rightarrow$ point$(\varphi_7)[0] \leftarrow T$

Instantiation 1: $s^{1*} = 5$:
sorted(s). *remove*(2), $\text{flip}_9[0] \leftarrow 2 - 1 = 1$ and $\text{flip}_9[1] = 3$.
We update the values in point(φ_9) that correspond to the signal values with positions 1 and 2 in sorted(s), that is, $s = 3$ and $s = 5$ corresponding to position indices 1 and 8 in point(φ_9).
point$(\varphi_9) = [F,T,T,T,T,T,T,F,F,T]$
intvl $(\varphi_9) = [2,7]$, $[10,10]$, intvl $(\varphi_8) = [1,2]$
Does intvl (φ_8) start with 1? Yes $\Rightarrow$ point$(\varphi_7)[1] \leftarrow T$

Instantiation 2: $s^{1*} = 7$:
sorted(s). *remove*(5), $\text{flip}_9[1] \leftarrow 3 - 1 = 2$ and $\text{flip}_9[2] = 5$.
We update the values in point(φ_9) that correspond to the signal values with positions 2,3 and 4 in sorted(s), that is, $s = 6$,$s = 7$ and $s = 7$ corresponding to position indices 7,2 and 10 in point(φ_9).

point(φ_9) = $[F, T, T, T, T, T, F, F, F, F]$
intvl (φ_9) = $[3, 7]$, intvl (φ_8) = $[2, 2]$
Does intvl (φ_8) start with 2? Yes $\Rightarrow$ point(φ_7)[2] $\leftarrow T$

By the end of all the instantiations of the freeze variable s^1, this is how point(φ_7) and intvl (φ_7) look like:
point(φ_7) = $[T, T, T, F, F, F, F, F, F, F]$, intvl ($\varphi_7$) = $[0, 2]$

Then the algorithm proceeds to the next iteration of the *for* loop in Line 4 of 4 and calculates intvl(φ_6), point(φ_5), intvl(φ_5), intvl(φ_3) for all the instantiations of s^2 to get the values of point(φ_2) and eventually intvl(φ_2).

Finally, it runs Lines 26-28 to calculate intvl (φ_1) and returns it.

7 RUNNING TIME: OPTIMIZED ALGORITHM

Before we give the complexity of our optimized algorithm, let us first introduce the following variables:

- $|\,\text{SubTree}(x_k)|$: number of sub-formulae in SubTree(x_k).
- V: the number of freeze variables in φ.
- $|\text{intvl}\,(\varphi)|$: maximal size of $start_j$ for any subformula in φ.
- $|\,\text{AST}(\varphi)|$: number of subformulae in φ.

We have the following complexities:

- Sorting a signal component (Line 3 of the optimized algorithm) takes $O\,(|\rho| \cdot \log(|\rho|))$
- `Transform` Algorithm: $O(|\rho|)$ (*for* loop in Line 3).
- `UpdateSignalConstraint` Algorithm: $O(|\rho|)$ (the Line 5 for loop has at most $|\rho|$ iterations. The Line 7 if condition makes sure Lines 8-12 have at most $O(|\rho|)$ complexity).
- `Sub-Update` Algorithm: $O(1)$.
- `ComputeIntervals` Algorithm: $O(|\text{intvl}\,(\varphi)|)$ for a uniformly sampled trace and $O\,(\min\,(|\rho|, |\text{intvl}\,(\varphi)| \cdot \log(|\rho|)))$ for non uniformly sampled trace (as explained in section 5.2.8).

Now, for our optimized algorithm, calculating intvl(φ_j) for all subformula φ_j (other than signal constraints) in SubTree(x_k) for just one instantiation (for loop in Line 16-20) takes $O(|\,\text{SubTree}(x_k)| \cdot |\text{intvl}(\varphi)|)$ time for a uniformly time-sampled trace and $O\,(\min\,(|\rho|, |\text{intvl}(\varphi)| \cdot \log(|\rho|)))$ for non-uniformly time-sampled trace. However, this is only when we already have intvl(φ_j) for the signal constraint φ_j (Line 18). Updating a signal constraint takes $O(|\rho|)$ for each instantiation.

To sum up, for each freeze variable x_k (Line 4), the algorithm needs to iterate over $|\rho|$ instantiations (Line 15), and for each instantiation, we need to update all the subformula in SubTree(x_k) including signal constraints (Lines 16-20). Thus, the complexity of the algorithm will be:

- $O\,(|V| \cdot |\rho| \cdot \max(|\rho|, |\,\text{AST}(\varphi)| \cdot |\text{intvl}\,(\varphi)|))$ for a uniformly sampled trace.
- $O\left(|V| \cdot |\rho| \cdot \max\left(\begin{array}{l} |\rho|, \\ |\,\text{AST}(\varphi)| \cdot \min(|\rho|, |\text{intvl}\,(\varphi)| \cdot \log|\rho|) \end{array}\right)\right)$ for a non-uniformly sampled trace.

For a non-uniformly sampled trace, updating the signal constraint takes the same time as in the uniform case – $O(|\rho|)$, the first term of the max expressions above. However, calling `ComputeIntervals` with each instantiation will take $O\,(\min\,(|\rho|, |\text{intvl}(\varphi)| \cdot \log(|\rho|)))$ instead of $O(|\text{intvl}(\varphi)|)$ which explains the higher complexity. Note that, in our complexity analysis, we assume that we have a constant number of signal constraints.

In practice, we expect $|\text{intvl}(\varphi)|$ to be very low compared to the trace size. We also expect that updating the signal constraint will not require $O(|\rho|)$ time. In fact, from one instantiation to the next one, only few values in point φ (where φ 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 the real world are continuous and do not have sudden value shifts.

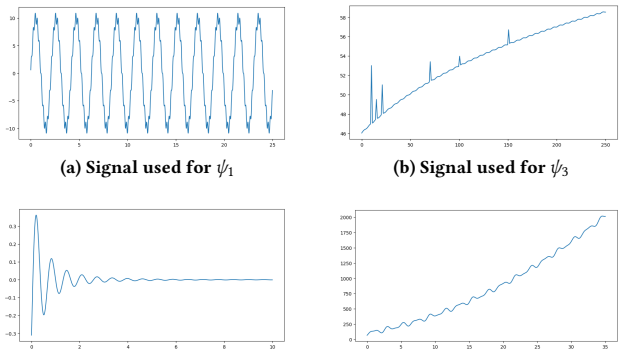
Overall, we would expect both factors inside the max to be low; and hence we expect sub-quadratic, $O(|\rho| \cdot \log(|\rho|))$ running times (in terms of trace size) for uniformly sampled traces (at least for formulae and traces with a low number of intervals), the $|\rho| \cdot \log(|\rho|)$ factor is from sorting the signal values (Line 3 of the algorithm). For the non-uniformly sampled case, we expect similar running times as the uniformly sampled trace since we similarly expect the min factor to be low as well.

8 EXPERIMENTS: OPTIMIZED ALGORITHM

Our experiments were conducted on a 64-bit Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz with 16-GB RAM and we implemented our optimized algorithm using C++. We generated the traces using Python. Noise was added to all traces by superimposing a noise signal. Some of the signals that we used in our experiments are shown in Figure 4. We use the formulae in section 3 in our experiments. In addition, we use the following formula to test our code for formulae with more than one freeze variable:

$$\psi_7 = s^1.\Diamond_{[0,\epsilon]}\left(s - s^{1*} > a \wedge s^2.(\Box_{[0,T]}s - s^{2*} \leq \delta)\right).$$

The obtained results are shown in Table 1 and conform to our



(a) Signal used for ψ_1 (b) Signal used for ψ_3

(c) Signal used for ψ_4 (d) Signal used for ψ_5

Figure 4: Some generated signals used in the experiments

complexity analysis. We can see that the size of $|\text{intvl}\,(\psi_i)|$ depends on the STL^*_1 requirement and the trace. For ψ_1 for example, the requirement is to check if the signal is oscillating with a period at most 10-time units, the bigger the trace, the more oscillations we need to check, the higher $|\text{intvl}\,(\psi_i)|$. Other formulae like ψ_2, ψ_5, ψ_6 and ψ_7 check conditions that do not scale with the trace, thus, $|\text{intvl}\,(\psi_i)|$ is constant for them.

In the non-uniformly sampled trace case, since our algorithm calls the *trim* function to iterate over the intervals of a given subformula, it can perform, in the best case, as well as in the uniform case . Or, in the worst case, the algorithm runs in quadratic time

	$\|\rho\| = 2500$			$\|\rho\| = 5000$			$\|\rho\| = 10000$			$\|\rho\| = 20000$			$\|\rho\| = 100000$		
ψ	A	B	n	A	B	n	A	B	n	A	B	n	A	B	n
ψ_1	0.086	0.091	25	0.242	0.264	49	0.891	0.921	97	3.394	4.576	193	85.78	89.56	957
ψ_2	0.024	0.024	1	0.035	0.036	1	0.055	0.053	1	0.097	0.101	1	0.503	0.514	1
ψ_3	0.038	0.045	11	0.061	0.086	11	0.110	0.153	11	0.216	0.314	11	1.109	2.104	11
ψ_4	0.052	0.055	40	0.101	0.107	80	0.278	0.292	159	0.564	0.593	318	1.071	1.209	1571
ψ_5	0.027	0.028	2	0.038	0.040	2	0.052	0.056	2	0.088	0.092	2	0.363	0.392	2
ψ_6	0.027	0.028	3	0.044	0.047	3	0.068	0.072	3	0.123	0.131	3	0.417	0.431	3
ψ_7	0.025	0.025	2	0.031	0.033	2	0.048	0.049	2	0.080	0.085	2	0.426	0.441	2

A: uniformly sampled trace monitoring times (seconds); B: non-uniformly sampled trace monitoring times (seconds); $n = |\text{intvl}(\psi_i)|$.

Table 1: Results for the different subformulae for different $|\rho|$ values

complexity in terms of trace size. In practice, in most cases, the algorithm runs in sub-quadratic time.

Formula ψ_3 checks the spike requirement, we ran experimenta with different numbers of spikes and we noted that each extra spike adds +1 to the number of intervals of the subformula $s.(\Diamond_{[0,w]}(s - s^* > \delta \wedge \Diamond_{[0,w]}|s - s^*| \leq \epsilon))$. The position of the spikes does not affect the running time for the non-uniformly sampled trace algorithm, or for the uniformly sampled trace case. In table 1, the trace that we used has 10 spikes.

For ψ_4, we recall that this formula checks if the settling time of a signal exceeds a certain value. The low running time can be explained by the fact that the shown high number of intervals is only within the first 150 instantiations (for all trace sizes), after that, the maximum value of $|\text{intvl}(\psi_4)|$ is 1. In fact, the average number of intervals per instantiation and per subformula is 3.37 for all trace sizes. The obtained running times show that our complexity is an overestimation and in practice, our algorithm's running time is affected by the average number of intervals and not the maximal number of intervals.

Note that, in our experiments, ψ_1 is the same formula (7) used in [9] and ψ_5 matches the template of formulae (8) and (9) from the same work. The authors in [9] claim that "a single run of the monitoring algorithm for the formulae (7), (8) or (9) over a signal sampled by 80 points took several hours on a regular PC" while our algorithm takes milliseconds for a trace of length 2500.

We conduct a second set of experiments on ψ_1. We use a trace that consists of a periodic signal and we fix the number of timestamps to $|\rho| = 10000$. We vary the signal frequency in order to change the $|\text{intvl}(\psi_1)|$ value. We report the obtained results in table 2. Again, the obtained results conform to our complexity analysis.

	$n = 25$	$n = 49$	$n = 97$	$n = 193$	$n = 383$
A	0.266	0.442	0.868	1.704	3.231
B	0.271	0.484	0.941	1.953	4.593

$n = |\text{intvl}(\psi_1)|$.

Table 2: Results for the different $|\text{intvl}(\psi_1)|$ values

9 CONCLUSION

In order to overcome the expressivity limitations of STL, researchers have investigated augmentations with additional temporal operators. Apart from [5, 9], the work in [4] which augments STL with max/min operators over windows, and presents a linear time monitoring procedure, is relevant. While their logic augments STL, it cannot express many of the properties expressible in STL_1^*.

Freeze quantification has been long studied [2, 8], and provides a natural syntax for expressing many commonly occurring engineering properties that cannot be expressed in STL. Our present work is the first one to show that monitoring still remains tractable with the addition of signal-value freeze quantification to STL (provided that freeze quantifiers are independent of each other). We provide experimental validation for our monitoring algorithms and demonstrate that our proposed algorithms scale to trace lengths of 100k. In addition to being the *first scalable monitoring algorithms* for signal-value freeze quantification, our algorithms do not use any specialized libraries such as those for manipulating polyhedra; and hence are efficiently *implementable* on a wide range of platforms.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation by a CAREER award (grant number 2240126).

REFERENCES

[1] R. Alur and D. L. Dill. 1994. A Theory of Timed Automata. *Theor. Comput. Sci.* 126, 2 (1994), 183–235.
[2] R. Alur and T. A. Henzinger. 1994. A Really Temporal Logic. *J. ACM* 41, 1 (1994), 181–204.
[3] C. Baier and J. P. Katoen. 2008. *Principles of model checking.* MIT Press.
[4] A. Bakhirkin and N. Basset. 2019. Specification and Efficient Monitoring Beyond STL *(LNCS, Vol. 11428).* Springer, 79–97.
[5] A. Bakhirkin, T. Ferrère, T. A. Henzinger, and D. Nickovic. 2018. The first-order logic of signals: keynote. In *EMSOFT'18.* IEEE, 1.
[6] E. Bartocci, J. V. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Nickovic, and S. Sankaranarayanan. 2018. Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications. In *Lectures on Runtime Verification - Introductory and Advanced Topics.* LNCS, Vol. 10457. Springer, 135–175.
[7] J. L. Bentley and A. C. Yao. 1976. An almost optimal algorithm for unbounded searching. *Inform. Process. Lett.* 5, 3 (1976), 82–87.
[8] P. Bouyer, F. Chevalier, and N. Markey. 2010. On the expressiveness of TPTL and MTL. *Inf. Comput.* 208, 2 (2010), 97–116.
[9] L. Brim, P. Dluhos, D. Safránek, and T. Vejpustek. 2014. STL*: Extending signal temporal logic with signal-value freezing operator. *Inf. Comput.* 236 (2014), 52–67.
[10] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia. 2017. Robust online monitoring of signal temporal logic. *Formal Methods Syst. Des.* 51, 1 (2017), 5–30.
[11] A. Dokhanchi, B. Hoxha, C.E. Tuncali, and G. Fainekos. 2016. An efficient algorithm for monitoring practical TPTL specifications. In *MEMOCODE'16.* IEEE, 184–193.
[12] A. Donzé. 2010. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In *CAV'10 (LNCS, Vol. 6174).* Springer, 167–170.
[13] A. Donzé, T. Ferrère, and O. Maler. 2013. Efficient Robust Monitoring for STL. In *CAV'13 (LNCS 8044).* Springer, 264–279.
[14] G. Ernst, S. Sedwards, Z. Zhang, and I. Hasuo. 2021. Falsification of Hybrid Systems Using Adaptive Probabilistic Search. *ACM Trans. Model. Comput. Simul.* 31, 3 (2021), 18:1–18:22.
[15] G. Fainekos, B. Hoxha, and S. Sankaranarayanan. 2019. Robustness of Specifications and Its Applications to Falsification, Parameter Mining, and Runtime Monitoring with S-TaLiRo. In *RV'19 (LNCS, Vol. 11757).* Springer, 27–47.

[16] G. E. Fainekos and G. J. Pappas. 2009. Robustness of temporal logic specifications for continuous-time signals. *Theor. Comput. Sci.* 410, 42 (2009), 4262–4291.
[17] G. E. Fainekos, S. Sankaranarayanan, K. Ueda, and H. Yazarel. 2012. Verification of automotive control applications using S-TaLiRo. In *ACC'12*. IEEE, 3567–3572.
[18] B. Ghorbel and V. S. Prabhu. 2022. Linear Time Monitoring for One Variable TPTL. In *HSCC '22*. ACM, 5:1–5:11.
[19] B. Ghorbel and V. S. Prabhu. 2023. Quantitative Robustness for Signal Temporal Logic with Time-Freeze Quantifiers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 12 (2023), 4436–4449.
[20] A. Grez, F. Mazowiecki, M. Pilipczuk, G. Puppis, and C. Riveros. 2021. Dynamic Data Structures for Timed Automata Acceptance. In *IPEC' 21 (LIPIcs, Vol. 214)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:18.
[21] J. Kapinski, X. Jin, J. Deshmukh, A. Donzé, T. Yamaguchi, H. Ito, T. Kaga, S. Kobuna, and S. Seshia. 2016. ST-Lib: A Library for Specifying and Classifying Model Behaviors. *SAE Technical Paper Series*.
[22] Z. Kong, A. Jones, and C. Belta. 2017. Temporal Logics for Learning and Detection of Anomalous Behavior. *IEEE Trans. Autom. Control.* 62, 3 (2017), 1210–1222.
[23] W. Liu, N. Mehdipour, and C. Belta. 2022. Recurrent Neural Network Controllers for Signal Temporal Logic Specifications Subject to Safety Constraints. *IEEE Control. Syst. Lett.* 6 (2022), 91–96.
[24] O. Maler and D. Nickovic. 2004. Monitoring Temporal Properties of Continuous Signals. In *FORMATS/FTRTFT*. Springer, 152–166.
[25] O. Maler and D. Nickovic. 2013. Monitoring properties of analog and mixed-signal circuits. *STTT* 15, 3 (2013), 247–268.
[26] N. Markey and J-F. Raskin. 2006. Model checking restricted sets of timed paths. *Theor. Comput. Sci.* 358, 2-3 (2006), 273–292.
[27] V. S. Prabhu and M. Savaliya. 2022. Towards Efficient Input Space Exploration for Falsification of Input Signal Class Augmented STL. In *MEMOCODE' 22*. IEEE, 1–11.
[28] V. Raman, A. Donzé, M. Maasoumy, R.M. Murray, A.L. Sangiovanni-Vincentelli, and S.A. Seshia. 2017. Model Predictive Control for Signal Temporal Logic Specification. *CoRR* abs/1703.09563 (2017). arXiv:1703.09563
[29] V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia. 2015. Reactive synthesis from signal temporal logic specifications. In *HSCC'15*. ACM, 239–248.
[30] G. Rosu and K. Havelund. 2001. *Synthesizing Dynamic Programming Algorithms From Linear Temporal Logic Formulae*. Technical Report.
[31] S. Sankaranarayanan, S. A. Kumar, F. Cameron, B. W. Bequette, G. Fainekos, and D.M. Maahs. 2017. Model-based falsification of an artificial pancreas control system. *SIGBED Rev.* 14, 2 (2017), 24–33.
[32] M. Waga and I. Hasuo. 2018. Moore-Machine Filtering for Timed and Untimed Pattern Matching. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 37, 11 (2018), 2649–2660.
[33] M. Waga, I. Hasuo, and K. Suenaga. 2017. Efficient Online Timed Pattern Matching by Automata-Based Skipping. In *FORMATS' 17, Proceedings (LNCS 10419)*. Springer, 224–243.