

STLCG++: A Masking Approach for Differentiable Signal Temporal Logic Specification

Parv Kapoor^{1,2}, Kazuki Mizuta³, Eunsuk Kang¹ and Karen Leung^{3,4}

Abstract—Signal Temporal Logic (STL) offers a concise yet expressive framework for specifying and reasoning about spatio-temporal behaviors of robotic systems. Attractively, STL admits the notion of *robustness*, the degree to which an input signal satisfies or violates an STL specification, thus providing a nuanced evaluation of system performance. In particular, the differentiability of STL robustness enables direct integration to robotic workflows that rely on gradient-based optimization, such as trajectory optimization and deep learning. However, existing approaches to evaluating and differentiating STL robustness rely on recurrent computations, which become inefficient with longer sequences, limiting their use in time-sensitive applications. In this paper, we present STLCG++, a masking-based approach that parallelizes STL robustness evaluation and backpropagation across timesteps, achieving significant speed-ups compared to a recurrent approach. We also introduce a smoothing technique to enable the differentiation of *time interval bounds*, thereby expanding STL’s applicability in gradient-based optimization tasks involving spatial and temporal variables. Finally, we demonstrate STLCG++’s benefits through three robotics use cases and provide JAX and PyTorch libraries for seamless integration into modern robotics workflows. Project website with demo and code: <https://uw-ctrl.github.io/stlcg/>.

Index Terms—Formal Methods in Robotics and Automation, Deep Learning Methods, Software Tools for Robot Programming

I. INTRODUCTION

MANY robot planning tasks hinge on meeting desired spatio-temporal requirements, like a quadrotor navigating specific regions within strict time windows. Signal temporal logic (STL) [1] presents an attractive formalism to describe spatio-temporal specifications as it is designed to operate over *real-valued* time-series input rather than discrete propositions. In particular, STL is equipped with *quantitative* semantics, or *robustness formulas*, which measure how well a given robot trajectory satisfies a requirement. With some smoothing approximations in place, it becomes efficient and stable to *differentiate* STL robustness within gradient-based optimization methods—the key to many robot control and learning applications. As such, we have seen a growing

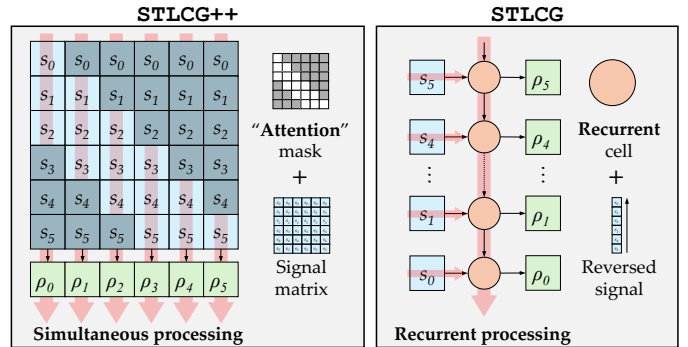


Fig. 1: We propose STLCG++, a masking approach to evaluating and backpropagating through STL robustness formulas. The masking approach offers stronger computational, theoretical, and practical benefits compared to STLCG, a recurrent approach.

interest in the inclusion of STL objectives/constraints in various optimization-based robotics problems utilizing gradient descent as a solution method, such as trajectory optimization [2], deep learning [3], and control synthesis [4].

Recently, STLCG [5] was introduced as a general framework for encoding any STL robustness formula as a computation graph, leveraging modern automatic differentiation (AD) libraries for evaluation and backpropagation. The STLCG (PyTorch) library democratized STL for robotics and deep learning communities, enabling recent work in gradient-based optimization with STL objectives/constraints [6]–[9].

To construct the computation graph for any STL robustness formula, STLCG processes the time-series input *recurrently* (see Fig. 1 right), primarily inspired by how recurrent neural networks (RNNs) [10] process sequential data. While consistent with the semantics of STL robustness, recurrent processing leads to the forward and backward passes being comparatively slower than other non-recurrent operations—a widely observed drawback of RNNs. These sequential operations limit STLCG’s capability for efficiently handling long sequence lengths in offline and online settings, especially when combined with other demanding computations, e.g., running foundation models. More recently, attention-based neural architectures, such as transformers [11], have demonstrated superior performance in processing sequential data, particularly on GPU hardware. The key to the transformer architecture is the self-attention operation, which operates on all input values *simultaneously* rather than recurrently.

Inspired by the masking mechanism in transformer architectures, we present STLCG++, a masking approach to evaluate and backpropagate through STL robustness for long sequences more efficiently than STLCG, a recurrent-based approach (see Fig. 1 left). STLCG++ opens new possibilities for using STL

Manuscript received: February 4th, 2025; Revised May 11th, 2025; Accepted June 23rd, 2025.

This paper was recommended for publication by Editor Lucia Pallottino upon evaluation of the Associate Editor and Reviewers’ comments. PK is supported by a National Science Foundation award under Grant No. 2144860. KM is supported by the UW+Amazon Science Hub Fellowship and Nakajima Foundation Fellowship.

¹Parv Kapoor and Eunsuk Kang are with School of Computer Science, Carnegie Mellon University {parvk, eunsukk}@andrew.cmu.edu

²Parv Kapoor is also affiliated with General Robotics

³Kazuki Mizuta and Karen Leung are with University of Washington, {mizuta, kymlleung}@uw.edu

⁴Karen Leung is also affiliated with NVIDIA

Digital Object Identifier (DOI): 10.1109/LRA.2025.3588389

TABLE I: Summary of existing python-based STL toolboxes that are publicly available.

Name	AutoDiff	Vectorize	GPU
STLGG++ (Ours)	JAX, PyTorch	✓	✓
STLGG [5]	PyTorch	✓	✓
Argus [15]	✗	✗	✗
stlpy [16]	✗	✗	✗
PyTeLo [17]	✗	✗	✗
pySTL [18]	✗	✗	✗
RTAMT [14]	✗	✗	✗

requirements in long-sequence contexts, especially for online computations, paving the way for further advancements in spatio-temporal behavior generation, control synthesis, and analysis for robotics applications.

Contributions. The contributions of this paper are fourfold. **(i)** We present STLGG++, a masking-based approach to computing STL robustness, and demonstrate the computational, mathematical, and practical benefits over STLGG, a recurrent-based approach. **(ii)** We apply a smooth masking mechanism to enable differentiation with respect to time interval parameters. To the best of the authors' knowledge, this is the first application of differentiating with respect to time parameters of STL formulas. **(iii)** We demonstrate the benefits of our proposed masking approaches with several robotics-related problems ranging from unsupervised learning, trajectory planning, and deep generative modeling. **(iv)** We provide *two* open-source STLGG++ libraries, one in JAX and another in PyTorch, and demonstrate their usage via the examples studied in this paper. JAX and PyTorch are well-supported Python libraries used extensively by the deep learning and optimization communities.

II. RELATED WORK

As STL has been used for various applications, a variety of STL libraries across different programming languages have been developed, including Python, C++, Rust, and Matlab. Given that Python is commonly used for robotics research, Tab. I compares recent STL Python packages regarding automatic differentiation (AD), vectorization, and GPU compatibility. Most libraries offer evaluation capabilities of a single signal, or their design is tailored towards a specific use case, making it difficult to extend or apply them to new settings. If users want to perform an optimization utilizing STL robustness formulas, a separate optimization package (e.g., CVXPY [12], Drake [13]) is often required.

RTAMT [14] was introduced as a unified tool for offline and online STL monitoring with an efficient C++ backend. It has received widespread support and has superseded other alternatives in terms of usage. However, RTAMT performs CPU-based signal evaluation and lacks differentiation and vectorization capabilities, limiting its efficiency in handling and optimizing over large datasets, where AD and GPU compatibility are crucial. STLGG was the first to introduce vectorized STL evaluation and backpropagation by leveraging modern AD libraries. However, STLGG faced scalability challenges due to its underlying recurrent computation. STLGG++ addresses these scalability limitations by eliminating recurrent operations and optimizing GPU capabilities.

III. PRELIMINARIES

We provide a brief introduction to STL and related terminologies. See [1], [5] for a more in-depth description.

A. Signals and trajectories

Signals and subsignals. STL formulas are interpreted over one-dimensional *signals* $\mathbf{s} = (s_0, \dots, s_T)$, a sequence of scalars sampled at uniform timesteps Δt (i.e., continuous-time outputs sampled at finite time intervals) from any system of interest. Given a signal $\mathbf{s} = (s_0, \dots, s_T)$, a *subsignal* is a contiguous fragment of a signal. By default, we assume a subsignal will start at timestep t and end at the last timestep T . We denote such a subsignal by $\mathbf{s}_t = (s_t, \dots, s_T)$. If a subsignal ends at a different timestep from T , we denote it by $\mathbf{s}_t^K = (s_t, \dots, s_K)$. *Note:* The absence of a sub(super)script on \mathbf{s} implies that the signal starts (ends) at timestep 0 (T).

States, trajectories, and subtrajectories. Given a system of interest, let $x_t \in \mathbb{R}^n$ denote the state at timestep t . Let $\mathbf{x} = (x_0, \dots, x_T)$ denote a sequence of states sampled at uniform time steps Δt . Similar to how subsignals are defined, we denote a subtrajectory by $\mathbf{x}_t = (x_t, \dots, x_T)$.

From trajectories to signals. In this work, we focus specifically on signals computed from a robot's trajectory. As we will see in the next section, core to any STL formula are predicates which are functions mapping state to a scalar value, $\mu : \mathbb{R}^n \rightarrow \mathbb{R}$, with $s_t = \mu(x_t)$, $t = 0, \dots, T$. A signal can represent, for example, a robot's forward speed.

B. Signal Temporal Logic: Syntax and Semantics

STL formulas are defined recursively according to the following grammar [1], [19],

$$\phi ::= \top \mid \mu_c \mid \neg\phi \mid \phi \wedge \psi \mid \phi \mathcal{U}_{[a,b]} \psi \quad (1)$$

True Predicate Not And Until

The grammar (1) describes a set of recursive operations that, when combined, can create a more complex formula. The time interval $[a, b]$ refers to timesteps rather than specific time values. When the time interval is dropped in the temporal operators, it defaults to the entire length of the input signal. Other commonly used logical connectives and temporal operators can be derived as follows: *Or* ($\phi \vee \psi := \neg(\neg\phi \wedge \neg\psi)$), *Eventually* ($\diamond_{[a,b]} \phi := \top \mathcal{U}_{[a,b]} \phi$) and *Always* ($\square_{[a,b]} \phi := \neg \diamond_{[a,b]} \neg\phi$).

A predicate $\mu_c : \mathbb{R}^n \rightarrow \mathbb{R}$ is a function that takes, for example, a robot state and outputs a scalar (e.g., speed). Then, given a state trajectory $\mathbf{x} = (x_0, \dots, x_T)$, $x_t \in \mathbb{R}^n$, we use the notation $\mathbf{x} \models \phi$ to denote that the trajectory \mathbf{x} satisfies ϕ according to the Boolean semantics (2).

Boolean Semantics

$$\begin{aligned} \mathbf{x} \models \mu_c &\Leftrightarrow \mu(x_0) > c \\ \mathbf{x} \models \neg\phi &\Leftrightarrow \neg(\mathbf{x} \models \phi) \\ \mathbf{x} \models \phi \wedge \psi &\Leftrightarrow (\mathbf{x} \models \phi) \wedge (\mathbf{x} \models \psi) \\ \mathbf{x} \models \diamond_{[a,b]} \phi &\Leftrightarrow \exists t \in [a, b] \text{ s.t. } \mathbf{x}_t \models \phi \\ \mathbf{x} \models \square_{[a,b]} \phi &\Leftrightarrow \forall t \in [a, b] \text{ s.t. } \mathbf{x}_t \models \phi \\ \mathbf{x} \models \phi \mathcal{U}_{[a,b]} \psi &\Leftrightarrow \exists t \in [a, b] \text{ s.t. } (\mathbf{x}_t \models \psi) \\ &\quad \wedge (\forall \tau \in [0, t], \mathbf{x}_\tau \models \phi) \end{aligned} \quad (2)$$

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

STL also admits a notion of *robustness*—the *quantitative semantics* in (3) describes how much a signal satisfies or violates a formula. Positive robustness values indicate satisfaction, while negative robustness values indicate violation.

Quantitative Semantics (Robustness Formulas)

$$\begin{aligned}
 \rho(\mathbf{x}, \top) &= \rho_{\max} && \text{where } \rho_{\max} > 0 \\
 \rho(\mathbf{x}, \mu_c) &= \mu(x_0) - c \\
 \rho(\mathbf{x}, \neg\phi) &= -\rho(\mathbf{x}, \phi) \\
 \rho(\mathbf{x}, \phi \wedge \psi) &= \min(\rho(\mathbf{x}, \phi), \rho(\mathbf{x}, \psi)) \\
 \rho(\mathbf{x}, \diamond_{[a,b]}\phi) &= \max_{t \in [a,b]} \rho(\mathbf{x}_t, \phi) \\
 \rho(\mathbf{x}, \square_{[a,b]}\phi) &= \min_{t \in [a,b]} \rho(\mathbf{x}_t, \phi) \\
 \rho(\mathbf{x}, \phi \mathcal{U}_{[a,b]}\psi) &= \max_{t \in [a,b]} \left\{ \min \left(\min_{\tau \in [0,t]} \rho(\mathbf{x}_\tau, \phi), \rho(\mathbf{x}_t, \psi) \right) \right\}. \tag{3}
 \end{aligned}$$

C. Smooth max / min approximation

It is often desirable to use STL robustness within gradient-based optimization (e.g., training deep neural networks or trajectory optimization). Since the robustness formulas consist of nested max / min operations, smooth approximations $\widetilde{\max}/\widetilde{\min}$ are often used to aid numerical stability. Two typical smooth approximations are softmax and logsumexp.

$$\begin{aligned}
 \widetilde{\max}_{\text{soft}}(\mathbf{x}) &= \sum_{i=1}^N \frac{x_i \exp^{\tau x_i}}{\sum_{j=1}^N \exp(\tau x_j)}, \\
 \widetilde{\max}_{\text{LSE}}(\mathbf{x}) &= \frac{1}{\tau} \log \sum_{i=1}^N \exp(\tau x_i), \quad \widetilde{\min}(\mathbf{x}) = -\widetilde{\max}(-\mathbf{x})
 \end{aligned}$$

The smoothness can be controlled by a temperature parameter τ ; as $\tau \rightarrow \infty$, the smooth approximations approach the true max / min value. For deep learning, common practice is to anneal the temperature value so that a more accurate smooth max / min is used towards the end of the optimization routine. Soundness of the smooth approximation may be an important consideration, depending on the specific application. Recent works proposed sound approximations [20] which STLGG++ can easily support. However, we view these efforts as complementary to this paper.

D. Robustness trace

As STL formulas are composed of nested operations, evaluating the robustness value necessitates first determining the robustness value for each subtrajectory and subformula. For instance, consider $\diamond\phi$ where ϕ is the subformula. Evaluating $\rho(\mathbf{x}, \diamond\phi)$ requires the robustness value of ϕ for all subtrajectories, i.e., $\rho(\mathbf{x}_t, \phi)$ for all t . Suppose that ϕ is another temporal formula, e.g., $\square\varphi$, then we would also require $\rho(\mathbf{x}_t, \varphi)$ for all t . Subsequently, we introduce the concept of a *robustness trace*.

Definition 1 (Robustness trace): Given a trajectory $\mathbf{x} = (x_0, \dots, x_T)$ and an STL formula ϕ , the *robustness trace* $\tau(\mathbf{x}, \phi)$ is a sequence of robustness values of ϕ for each subtrajectory of \mathbf{x} . That is,

$$\tau(\mathbf{x}, \phi) = \rho(\mathbf{x}_0, \phi), \rho(\mathbf{x}_1, \phi), \dots, \rho(\mathbf{x}_T, \phi). \tag{4}$$

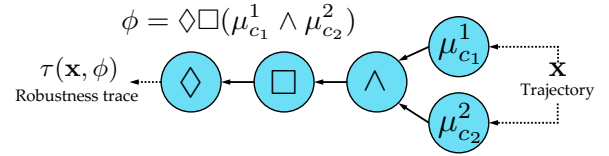


Fig. 2: Illustration of the graph structure of an example STL formula. Input trajectories are first passed through the predicates and then each subformula according to the formula structure.

Computing the robustness trace of non-temporal operations is straightforward, as there is no need to loop through time, and for brevity, we omit that description in this paper.

E. Recursive structure of STL formulas

To compute the robustness value of any STL formula ϕ with arbitrary formula depth¹, we must first calculate the robustness trace of its subformula(s), and so forth, since STL formulas are constructed recursively. Fig. 2 illustrates how the trajectories are passed through each operation of an STL formula to compute the robustness trace of each subformula. As such, we need to develop the “building blocks” for computing robustness corresponding to each STL operation. Then, by stacking these operations according to the formula structure, we can compute the robustness trace for *any* STL formula regardless of formula depth.

F. STLGG: Recurrent computation

We briefly outline the recurrent operations underlying STLGG; for more details, we refer the reader to [5]. Illustrated in Fig. 1 (right), STLGG utilizes the concept of dynamic programming to calculate the robustness trace. The input signal is processed *backward* in time, and a *hidden state* is maintained to store the information necessary for each recurrent operation at each time step. The choice of recurrent operation depends on the temporal STL formula (either a max or min). The size of the hidden state depends on the time interval of the temporal operator and is, at most, the length of the signal. Although the use of a hidden state to summarize past information helps reduce space complexity, the recurrent operation leads to slow evaluation and backpropagation due to sequential dependencies. Next, we present a masking-based approach that bypasses the sequential dependency, leading to faster computation times.

IV. MASKING APPROACH FOR TEMPORAL OPERATIONS

We propose STLGG++, a *masking* approach to compute STL robustness traces. Fig. 1 illustrates the masking approach, highlighting the idea that each value of a robustness trace is computed *simultaneously*, rather than sequentially, as we saw with STLGG. Mirroring the concept of attention masks in transformer architectures [11], we introduce a mask \mathcal{M} to select relevant parts of a signal that can be later processed simultaneously rather than sequentially. In the rest of this section, we describe the masking operation underpinning STLGG++ and several properties.

¹Assuming the formula consists of at least one temporal operation. Although STL formulas without temporal operators are possible, it is the temporal operations that make computing robustness challenging.

A. Notation and masking operation

Assume we have a 1D input signal \mathbf{s} of length $T+1$, and let K denote the number of time steps contained (inclusive) in the time interval $[a, b]$ given an STL temporal operator. A mask \mathcal{M} is a 2 or 3 dimensional array, depending on the temporal operator, whose entries are either 1 or 0. A mask \mathcal{M} is applied via element-wise multiplication to an array \mathcal{S} of the same size. E.g., for a 2D mask, if $\mathcal{M}_{ij} = 0$, the corresponding entries \mathcal{S}_{ij} will be ignored, or is *masked out*. We also replace the entries masked out with an arbitrary value, typically a large positive or negative value. Let $\mathcal{M} \odot_{\pm} \mathcal{S}$ denote the application of mask \mathcal{M} on \mathcal{S} and masked out by either a large positive (\odot_+) or negative (\odot_-) number.

B. Eventually and Always operations

For brevity, we describe the procedure for the Eventually operation but note that the procedure for the Always operation is almost identical except a min operation is used instead of max. Consider the formula $\psi = \diamond_{[a,b]} \phi$, and let $\mathbf{s} = \tau(\mathbf{x}, \phi)$ denote the robustness trace of ϕ . Specifically, $s_t = \rho(\mathbf{x}_t, \phi)$. Here, ϕ is an STL subformula, typically representing a predicate or a more complex temporal logic expression. Then, we seek to find the robustness trace,

$$\tau(\mathbf{x}, \diamond_{[a,b]} \phi) = \left\{ \max_{i \in [a,b]} s_{t+i} \right\}_{t=0}^T$$

Intuitively, we want to slide a time window $[a, b]$ along \mathbf{s} and take the max of the values within the window. Rather than performing this set of operations sequentially, we **first** “unroll” the sequential operation—turning the 1D signal into a 2D array, **second**, we apply a 2D mask to mask out the irrelevant entries dictated by the time window, and **third**, we use the max operation over the unmasked values.

Step 1: “Unrolling” the signal. First, we *pad* the end of the signal by the size of the time window K ; the reason will be apparent in Step 3. The padding value \bar{s} can be set by the user, such as extending the last value of the signal, or setting it to a large negative number. Then, we turn the padded signal into a 2D array \mathcal{S} by repeating the signal along the second dimension T times, resulting in a $(T+K) \times T$ 2D array. An example is provided in (5).

Step 2: Construct 2D mask. We construct a mask \mathcal{M} that will be applied to the unrolled signal \mathcal{S} . The mask \mathcal{M} comprises of two sub-masks: (i) subsignal mask $\mathcal{M}_{\text{subsig}}$ and (ii) time interval mask $\mathcal{M}_{\text{time}}$.

Subsignal mask $\mathcal{M}_{\text{subsig}}$: This mask incrementally masks out the start of the signal as we move along the horizontal (second) dimension of \mathcal{S} . Essentially, $\mathcal{M}_{\text{subsig}}$ masks out the upper triangular region with an offset of one to exclude the diagonal entries. After applying the $\mathcal{M}_{\text{subsig}}$ on \mathcal{S} , the columns of the unmasked entries correspond to all the subsignals of \mathbf{s} . The entries masked out by $\mathcal{M}_{\text{subsig}}$ are denoted by the red shaded entries in (5).

Time interval mask $\mathcal{M}_{\text{time}}$: This mask masks out all the entries outside of time interval $[a, b]$ for each subsignal. $\mathcal{M}_{\text{time}}$ consists of an off-diagonal strip that masks out entries *before* the start of the interval (determined by a), and a lower

triangular matrix with an offset that masks out entries *after* the interval (determined by b). The entries masked out by $\mathcal{M}_{\text{time}}$ are denoted by the blue shaded entries in (5).

Final mask \mathcal{M} : The final mask is $\mathcal{M} = \mathcal{M}_{\text{subsig}} + \mathcal{M}_{\text{time}}$, resulting in a mask that retains all the entries within the time interval $[a, b]$ for each sub-signal (unshaded entries in (5)).

Step 3: Apply 2D mask and max operation. Given \mathcal{S} and \mathcal{M} from the previous steps, we can compute $\mathcal{M} \odot_- \mathcal{S}$, similar to what is done in the mask attention mechanism in transformer architectures. Then, we apply the max operation along each column. Note the padding we did back in step 1 becomes relevant when the time interval exceeds the length of the subsignal. This is similar to the approach taken in [5] to handle incomplete signals. For the Always operation, min and $\mathcal{M} \odot_+ \mathcal{S}$ are used instead.

Example 1: Consider the STL formula $\psi = \diamond_{[1,3]}(s > 0)$ and a signal with 8 time steps, $\mathbf{s} = [s_0, \dots, s_7]$. The corresponding unrolled 2D array is shown (5) with padding value \bar{s} , and the red and blue shaded entries denote the entries masked out by $\mathcal{M}_{\text{subsig}}$ and $\mathcal{M}_{\text{time}}$, respectively.

$$\begin{bmatrix} s_0 & s_0 & s_0 & s_0 & s_0 & s_0 & s_0 & s_0 \\ s_1 & s_1 & s_1 & s_1 & s_1 & s_1 & s_1 & s_1 \\ s_2 & s_2 & s_2 & s_2 & s_2 & s_2 & s_2 & s_2 \\ s_3 & s_3 & s_3 & s_3 & s_3 & s_3 & s_3 & s_3 \\ s_4 & s_4 & s_4 & s_4 & s_4 & s_4 & s_4 & s_4 \\ s_5 & s_5 & s_5 & s_5 & s_5 & s_5 & s_5 & s_5 \\ s_6 & s_6 & s_6 & s_6 & s_6 & s_6 & s_6 & s_6 \\ s_7 & s_7 & s_7 & s_7 & s_7 & s_7 & s_7 & s_7 \\ \bar{s} & \bar{s} & \bar{s} & \bar{s} & \bar{s} & \bar{s} & \bar{s} & \bar{s} \\ \bar{s} & \bar{s} & \bar{s} & \bar{s} & \bar{s} & \bar{s} & \bar{s} & \bar{s} \\ \bar{s} & \bar{s} & \bar{s} & \bar{s} & \bar{s} & \bar{s} & \bar{s} & \bar{s} \end{bmatrix} \quad (5)$$

After filling the masked entries with a large negative value, and taking the max column-wise. Let $\mathbf{s} = [0, 1, 2, 3, 4, 5, 6, 7]$ and $\bar{s} = 7$, then $\tau(\mathbf{s}, \psi) = [3, 4, 5, 6, 7, 7, 7, 7]$. If we set $\bar{s} = -10^5$, then $\tau(\mathbf{s}, \psi) = [3, 4, 5, 6, 7, -10^5, -10^5, -10^5]$.

C. Until operation

We apply a similar masking strategy for the Until operation, following a similar three-step process described in Sec. IV-B. The main differences are (i) the signal needs to be “unrolled” into three dimensions, and (ii) the operations in Step 3 correspond to the Until robustness formula.

Consider the formula $\phi \mathcal{U}_{[a,b]} \psi$ where ϕ and ψ are STL subformulas. Let $\mathbf{s}^\phi = \tau(\mathbf{x}, \phi)$ and $\mathbf{s}^\psi = \tau(\mathbf{x}, \psi)$ denote the robustness trace of ϕ and ψ respectively for trajectory \mathbf{x} . Specifically, $s_t^\phi = \rho(\mathbf{x}_t, \phi)$ and $s_t^\psi = \rho(\mathbf{x}_t, \psi)$. Then, we seek to find the robustness trace,

$$\tau(\mathbf{x}, \phi \mathcal{U}_{[a,b]} \psi) = \left\{ \max_{i \in [a,b]} \left\{ \min \left(\min_{\tau \in [0,i]} s_{t+\tau}^\phi, s_{t+i}^\psi \right) \right\} \right\}_{t=0}^T$$

Step 1: “Unrolling” the signal. Given a signal \mathbf{s} , we construct a 3D array by repeating the signal along the first and second dimensions, resulting in a $(T+K) \times T \times K$ array. This unrolling operation on \mathbf{s}^ϕ and \mathbf{s}^ψ results in \mathcal{S}^ϕ and \mathcal{S}^ψ .

Step 2: Construct 3D mask. In the Until robustness formula, we need $\min_{\tau \in [0,i]} s_{t+\tau}^\phi$ and s_{t+i}^ψ for $i \in [a, b]$ and for all t . Given i , notice that $\rho(\mathbf{x}_t, \square_{[0,i]} \phi) = \min_{\tau \in [0,i]} s_{t+\tau}^\phi$ and $\rho(\mathbf{x}_t, \square_{[i,i]} \psi) = s_{t+i}^\psi$ which we can compute given the

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

TABLE II: Summary of STL specifications used for evaluation inspired by existing literature. I denotes the time interval over which the temporal operators apply.

Spec	Depth	Description	STL Formula
ϕ_1	0	Invariance	$\Box(\varphi \wedge \psi)$
ϕ_2	1	Stabilization	$\Diamond\Box(\varphi \wedge \psi)$
ϕ_3	1	Strict ordering	$\varphi \mathcal{U} \psi$
ϕ_4	3	Sequenced visit pattern	$\Diamond_I(\varphi_4 \wedge \psi_4 \wedge \Diamond_I(\varphi_3 \wedge \psi_3 \wedge \Diamond_I(\varphi_2 \wedge \psi_2 \wedge \Diamond_I(\varphi \wedge \psi))))$
ϕ_5	2	Sequenced visit + stabilization	$\Diamond_I((\varphi_2 \wedge \psi_2) \wedge \Diamond_I(\Box_I(\varphi \wedge \psi)))$
ϕ_6	0	Reach regions in any order	$\Diamond_I(\varphi_9 \wedge \psi_9) \wedge \dots \wedge \Diamond_I(\varphi_0 \wedge \psi_0)$

procedure outlined in Sec. IV-B. However, the outer max requires us to consider all $i \in [a, b]$. Thus for each formula $\Box_{[0,i]}\phi$ and $\Box_{[i,i]}\psi$, we stack all the 2D masks for each value of $i \in [a, b]$ across the third dimension, resulting in two 3D masks, \mathcal{M}^ϕ and \mathcal{M}^ψ of size $(T + K) \times T \times K$.

Step 3: Apply the Until robustness operations. We can now apply all the operations needed to compute the robustness trace for the Until operation. After Step 2, we have \mathcal{S}^ϕ , \mathcal{S}^ψ , and the 3D masks \mathcal{M}^ϕ and \mathcal{M}^ψ . We can compute $\mathcal{M}^\phi \odot_+ \mathcal{S}^\phi$ and $\mathcal{M}^\psi \odot_+ \mathcal{S}^\psi$. Then, we can apply the following set of operations (where the first dimension = 1).

$$\begin{aligned}
 S_1 &= \overbrace{\text{stack}_{\text{dim}=1} \left(\left[\min_{\text{dim}=1} \mathcal{M}^\phi \odot_+ \mathcal{S}^\phi, \min_{\text{dim}=1} \mathcal{M}^\psi \odot_+ \mathcal{S}^\psi \right] \right)}^{\text{Size } [2 \times T \times K]} \\
 S_2 &= \min_{\text{dim}=1} S_1, \quad (S_2 \text{ has size } [T \times K]) \\
 S_3 &= \max_{\text{dim}=2} S_2, \quad (S_3 \text{ has size } [T])
 \end{aligned}$$

We see that this set of operations is relatively simple, a series of min and max operations along various dimensions, and these operations remain the same regardless of the choice of time intervals, unlike the recurrent approach.

D. Dependence on smooth approximations

Both masking and recurrent approaches output the correct robustness values when the true min/max operations and logsumexp approximation are used. However, if the softmax/softmin approximation is used with a recurrent approach, the output robustness and gradient values do not correctly reflect the correct values. This is because, when recurrently applying softmin/softmax, values applied earlier in the recurrence will be “softened” more than values applied later in the recurrence. Not only does nesting the softmin/softmax operations give a very poor approximation of the true min/max value, but it also artificially reduces the gradient for values passed earlier in the recurrence. With the logsumexp approximation, applying it recursively over multiple values is the same as applying it once over all the values. That is, $\widehat{\text{max}}_{\text{LSE}}(\widehat{\text{max}}_{\text{LSE}}(\mathbf{x}), y) = \widehat{\text{max}}_{\text{LSE}}([\mathbf{x}, y])$ where as $\widehat{\text{max}}_{\text{soft}}(\widehat{\text{max}}_{\text{soft}}(\mathbf{x}), y) \neq \widehat{\text{max}}_{\text{soft}}([\mathbf{x}, y])$. This issue is not observed when using the masking-based approach due to the way subsignals are processed without recurrence.

E. Practical benefits

STL_{CG++} offers several practical benefits over STL_{CG}. We highlight several reasons why this is the case. **(i) Software design.** The recurrent approach required signals to be passed *backward* in time. This increased the complexity of software design and was not often intuitive for end users.

With the masking approach, there is no need to reverse the signal. **(ii) Static graph structure.** With the recurrent approach, the choice of time interval would fundamentally change the dimension of hidden states, whereas the graph structure remains the same for the masking approach. Keeping the same graph structure, even if the intervals change, is essential in compilation. **(iii) Vectorization and just-in-time compilation.** Related to the second point above, the static graph structure afforded by the masking approach enables the ability to easily *vectorize* and *just-in-time compile* (JIT) the computation over not just various signal inputs but also various time intervals. The ability to vectorize and JIT over different time intervals can be particularly useful in formula mining applications where we may want to evaluate robustness formulas with different time intervals simultaneously. **(iv) Conformity to popular deep learning libraries.** By leveraging AD libraries that empower popular frameworks, STL_{CG++} becomes accessible to the broader community and seamlessly integrates into existing tools.

V. COMPUTATIONAL PROPERTIES OF STL_{CG++}

In this section, we analyze the computational properties of the approaches STL_{CG++} (masking-based) and STL_{CG} (recurrent-based) by measuring the computation time required to compute robustness values and their gradient. We seek to answer the following research questions.

RQ1: Does STL_{CG++} compute robustness traces faster than STL_{CG} as measured by median computation time?

RQ2: How does STL_{CG++}’s computation time scale with sequence length compared to STL_{CG}?

We perform experiments on CPU and GPU. Since STL_{CG++} (masking) involves large matrix computations, we anticipate STL_{CG++} to scale favorably on a GPU. As STL_{CG} (recurrent) utilizes a recurrent structure, it scales with sequence length. Tab. II describes six different STL formulas with varying complexities [21] that we test on. We evaluate computation time for increasing signal lengths up to $T = 512$ time steps with a batch size of 8. We present our results in Fig. 3 and Tab. III and make the following observations.

CPU backend. We observe that STL_{CG++} generally achieves lower computation times than STL_{CG}. The exception is in ϕ_3 , where STL_{CG++} is slower than STL_{CG} for longer sequences. This is because the space complexity for the Until operation is $\mathcal{O}(T^3)$. In Tab. III, we see that STL_{CG++} on JAX struggled with ϕ_3 for long sequence lengths, but was fine with PyTorch. We hypothesize that it is due to how JAX allocates memory, especially during just-in-time compilations. Additionally, both libraries see increasing computation time

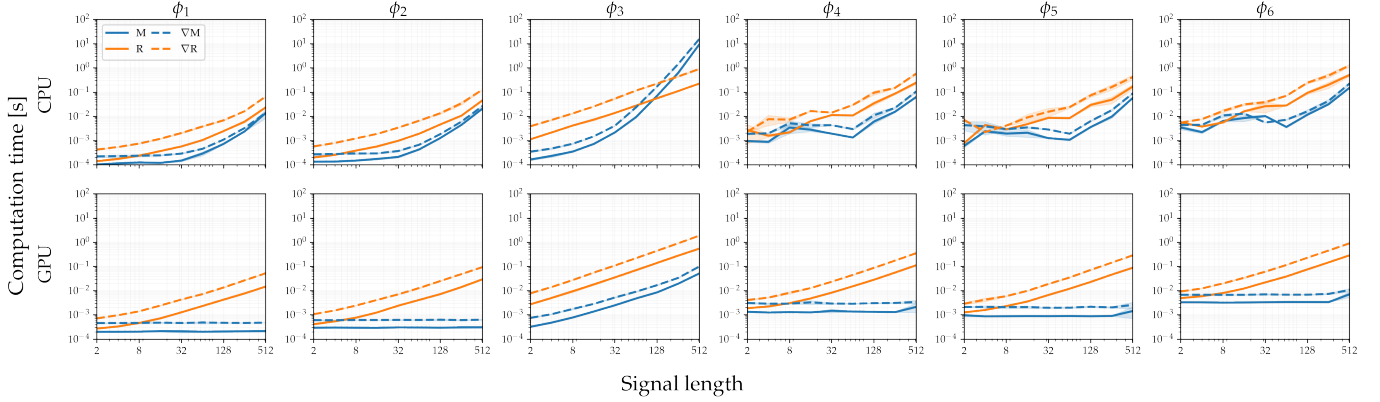


Fig. 3: Comparison of computation time using PyTorch for masking (M, blue) and recurrent (R, orange) approach on CPU (top row) and GPU (bottom row) as signal length increases, across six STL formulas. Solid lines denote robustness computation; dashed lines denote gradient evaluation. Results using JAX can be found on the website.

TABLE III: Relative computation time of masking approach compared to recurrent approach. Median value across different signal lengths. Lower is better.

Device	CPU	GPU	CPU	GPU
Formula	JAX + JIT		PyTorch	
ϕ_1	-37.99%	-93.43%	-76.05%	-85.76%
ϕ_2	-51.51%	-96.65%	-82.50%	-89.03%
ϕ_3	822.01%	-91.31%	-88.22%	-94.15%
ϕ_4	-68.22%	-96.00%	-73.48%	-77.91%
ϕ_5	-68.58%	-95.32%	-67.91%	-80.86%
ϕ_6	-72.93%	-98.11%	-54.14%	-84.36%

with sequence length. For STLCG the computation time increase is expected since the time complexity is $\mathcal{O}(T)$. For STLCG++, the increased computation time can be explained by the space complexity $\mathcal{O}(T^2)$ (except for ϕ_3), and such computations are not handled as efficiently on a CPU.

GPU backend. STLCG++ outperforms STLCG for all formulas. Additionally, STLCG++ exhibits essentially constant computation time except for ϕ_3 , which has $\mathcal{O}(T^3)$ space complexity. From Tab. III, we see that **STLCG++ on GPU provides around 95% and 85% improvement on JAX and PyTorch, respectively.** For STLCG, moving from CPU to GPU gives virtually the same scaling, with no significant improvement/reduction in computation times.

VI. SMOOTHING TIME INTERVALS

By using the masking operation to capture parts of the signal within the specified time interval, we can build a smooth approximation of the mask and *differentiate* robustness values with respect to the parameters of the mask that determine which values are selected. As such, it becomes possible to perform gradient descent over time interval parameters to find, for example, a time interval that best explains time-series data. We showcase two examples in Sec. VII utilizing this new capability. Notably, this differentiation was not possible with STLCG since the choice of time intervals impacted the size of the hidden state used within the recurrent computations.

Rather than using a mask with 0 and 1's as described in Sec. IV-B and IV-C, we use the following smooth mask approximation. For a sequence length of T , we have,

$$\text{mask}(i; a, b, c, \epsilon) = \max(\sigma(c(i - aT)) - \sigma(c(i - bT)) - \epsilon, 0), \quad (6)$$

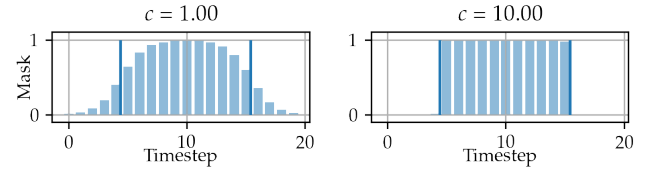


Fig. 4: Smooth mask for various values of smoothing parameter c . where i is the time index, $\sigma(x)$ is the sigmoid function, the parameters $0 \leq a < b \leq 1$ denote the fraction along the signal of the start and end of the time interval, c denotes the mask smoothing parameter, and ϵ denotes a user-defined tolerance. Fig. 4 shows a visualization of the smooth mask and how the smoothing parameter c affects the mask.

When optimizing over time interval parameters, we can anneal the smooth mask parameter c to help with convergence to a (local) optima on a generally nonlinear loss landscape (see Sec. VII-C for more details). As noted before, a benefit of the masking approach is that the operations can be easily vectorized. We can simultaneously evaluate the robustness for multiple time intervals and perform gradient descent on multiple values. As such, we can envision a use case where we perform a coarse global search via sampling and then a local refinement via gradient descent.

For reference, using the vectorized mapping function in JAX+JIT, the computation time to evaluate 90,000 different values for (a, b) with a signal of length $T = 20$ for the formula $\square_{[aT, bT]}(s > 0)$ is $1.87 \pm 0.0164\text{ms}$ on a M2 MacBookPro. This averages to around $20.78\mu\text{s}$ per time interval value. In contrast, (sequentially) searching over all possible valid time intervals for a signal length of 20 (i.e., 190 intervals with integer interval limits) using the recurrent approach takes about $5.25 \pm 0.0497\text{s}$, or roughly 27.6ms each evaluation. This means that **STLCG++ offers more than 1000 \times improvement in computation time** when evaluating STL robustness over multiple time interval values.

VII. ROBOTICS-RELATED APPLICATIONS

We showcase a variety of use cases that demonstrate the computational advantages of the masking approach. We developed *two* Python STLCG++ libraries, one in JAX [22] and

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

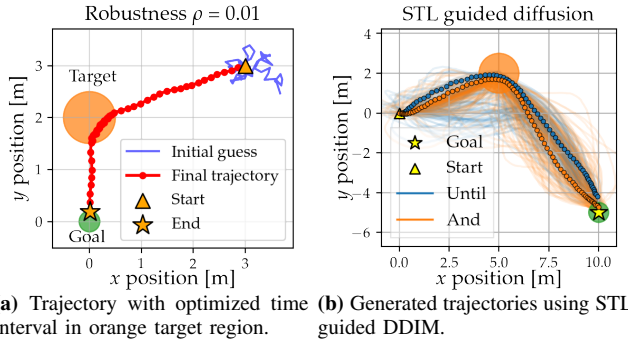


Fig. 5: Trajectories generated using STL_{CG++} with STL robustness in the planning objective.

another in PyTorch [23] and discuss how STL_{CG++} opens up new possibilities for robot planning and control.

A. Trajectory planning with suboptimal STL specifications.

Trajectory optimization is essential in robotic systems, enabling precise navigation and task execution under complex constraints. These trajectories must balance user-defined goals with feasibility and efficiency. Since goal misspecification can lead to erroneous behavior, infeasibility, or subpar performance, we show how suboptimal goals (captured through STL specifications) can be refined in addition to solving for an optimal trajectory. Consider a trajectory optimization problem where we would like to reach a goal region \bullet while visiting a target region \bullet within a fixed time horizon. Additionally, the visit duration of \bullet is to be maximized. Consider a specification $\phi = \square_{[aT, bT]}(\text{inside } \bullet) \wedge \diamond(\text{inside } \bullet)$ which specifies a time window that the trajectory should be inside \bullet . We cast this trajectory planning problem as an unconstrained optimization problem,

$$\begin{aligned} \min_{\mathbf{u}, a, b} \quad & \gamma_1 J_{\text{STL}}(\mathbf{x}) + \gamma_2 J_1(a, b) + \gamma_3 J_{\text{lim}}(\mathbf{u}) + \gamma_4 J_{\text{eff}}(\mathbf{u}), \\ J_{\text{STL}}(\mathbf{x}) = & \text{ReLU}(-\rho(\mathbf{x}, \phi)), \quad J_1(a, b) = \exp(2(\tilde{I} - b + a)), \\ J_{\text{lim}}(\mathbf{u}) = & \frac{1}{T} \sum_{t=0}^{T-1} \text{ReLU}(\|\mathbf{u}_t\|_2 - \bar{u}), \quad J_{\text{eff}}(\mathbf{u}) = \frac{1}{T} \sum_{t=0}^{T-1} \|\mathbf{u}_t\|_2^2, \end{aligned}$$

where \mathbf{x} is the state trajectory from executing \mathbf{u} with single integrator discrete-time dynamics. We use a timestep of $\Delta t = 0.1$, $T = 51$, $\bar{u} = 2$ is the system's maximum control limits, and $\tilde{I} = 0.2$ is a nominal (normalized) time interval size that we would like to improve upon. We randomly initialized the control inputs and $(a, b)_{\text{init}} = (0.14, 0.82)$. Using coefficients $\gamma_1 = 1.1, \gamma_2 = 0.05, \gamma_3 = 2, \gamma_4 = 0.5$, the resulting solution is shown in Fig. 5a with final values $(a, b)_{\text{final}} = (0.37, 0.84)$. The final optimized STL formula is $\phi = \square_{[20, 43]}(\text{inside } \bullet) \wedge \diamond(\text{inside } \bullet)$. Using JAX, each gradient step took about $92.3 \pm 3.99 \mu\text{s}$ on an M2 MacBookPro. With unconstrained multiobjective optimization, selecting objective weights that achieve desirable behavior can be tedious. With STL_{CG++}, we can perform gradient descent computation over various coefficient values simultaneously and then select the best one.

B. Deep generative modeling: STL-guided diffusion policies

We demonstrate the use of STL specifications for guided diffusion models [24], a recent type of deep generative models

that present a promising approach for robot policy learning and behavior generation [25]. We build upon [26], which uses Control Barrier and Lyapunov functions (CBFs and CLFs) to guide the denoising process of a diffusion model for safe control sequence generation. In this application, we use STL robustness in place of CBFs and CLFs as the guidance function. We trained a Denoising Diffusion Implicit Model (DDIM) model [24], generating trajectories with 80 time steps.

We used 200 denoising steps and a batch size of 128. We generated trajectory samples using two different STL guidance functions,

$$\begin{aligned} \phi_U &= \diamond_{[0, 10]}(\text{inside } \bullet) \cup \diamond_{[0, 5]}(\text{inside } \bullet) \\ \phi_\wedge &= \diamond_{[0, 10]}(\text{inside } \bullet) \wedge \diamond_{[60, 80]} \square_{[0, 5]}(\text{inside } \bullet). \end{aligned}$$

The resulting trajectories are shown in Fig. 5b. Although guided diffusion does not guarantee that the STL specifications are strictly satisfied, we observe a satisfaction rate of 52.34%, 62.50% for ϕ_U and ϕ_\wedge , respectively. This shows how STL_{CG++} can enhance diffusion policies to promote safer behavior generation.

C. Machine learning: STL parameter mining from data

Formal STL specifications play a crucial role in building robust robot systems, contributing to controller synthesis [27], fault localization [28], and anomaly detection and resolution [29]. However, specifications are not often readily available, making the task of mining specifications from data a vital problem to study [21]. In this example, we perform STL specification mining from data. Specifically, we leverage the differentiability and vectorization over smooth time intervals to find a time interval that best fits the observed data.

Consider a dataset of (noisy) signals with $T = 20$ timesteps. The signals have value 1 between the normalized time interval of $(a, b) = (0.23, 0.59)$ and zero elsewhere. Some noise is added around (a, b) and to the signal itself. The goal is to learn the largest time interval $[\bar{a}, \bar{b}]$ such that the signals from the dataset satisfy $\phi = \square_{[\bar{a}, \bar{b}]}(s > 0)$. We frame this STL mining problem as an optimization problem,

$$\min_{0 \leq a < b \leq 1} \frac{1}{N} \sum_{s \in \mathcal{D}} \max(-\rho(s, \square_{[aT, bT]}(s > 0)), 0) + \gamma(a - b) \quad (7)$$

where γ is a coefficient on the term $a - b$ that encourages the interval to be larger. We solve (7) via gradient descent, using the smooth time interval mask discussed in Sec. VI. Specifically, we annealed the time interval mask scaling factor and the temperature with a sigmoid schedule, and performed 5000 gradient steps with a step size of 10^{-2} . To ensure $0 \leq a < b \leq 1$, we passed them through a sigmoid function first. Fig. 6 shows a few snapshots of the gradient steps and the loss landscape as the scaling parameter and temperature increase. We can observe that our solution converges to the global optimum and is consistent with the ground truth values (subject to the injected noise). To give a sense of the computation time, using JAX on a M2 MacbookPro, it took about $122 \pm 0.479 \mu\text{s}$ per gradient step.

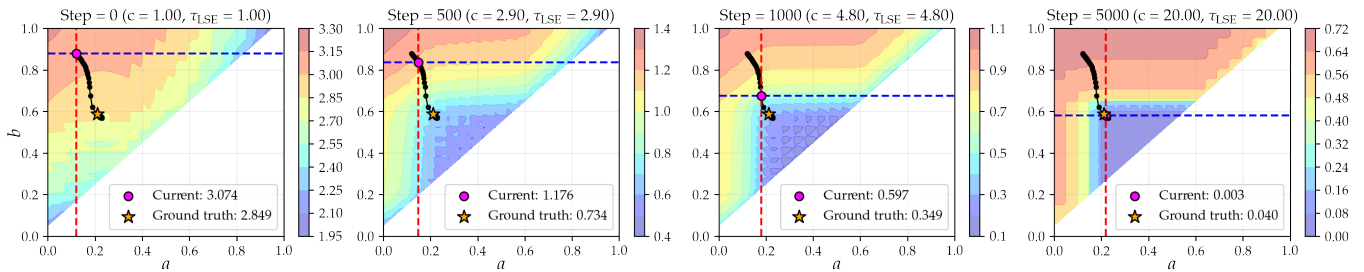


Fig. 6: The masking approach enables a smooth approximation of time intervals, enabling the use of gradient descent over time interval parameters. The contour plots show the loss landscape given by (7) as a function of a, b , the normalized time interval limits. The solutions at gradient descent steps 0, 500, 1000, and 5000 are shown. Note that the smooth time mask parameter and approximate max / min temperature values are annealed over the steps.

VIII. CONCLUSION

We present STLGG++, a masking-based approach for computing STL robustness using automatic differentiation libraries. STLGG++ mimics the operations that underpin transformer architectures, and we demonstrate the computational, theoretical, and practical benefits of the proposed masking approach over STLGG, which uses a recurrent approach. The observed advantages of masking over recurrent operations mirror the advantages of using transformers over recurrent neural networks for processing sequential data. We also present two STLGG++ libraries in JAX and PyTorch, demonstrating their usage in several robotics-related problems such as machine learning, trajectory planning, and deep generative modeling. STLGG++ offers significant computational advantages over STLGG, thus presenting new and exciting opportunities for incorporating STL specifications into various online robot planning and control tasks that require fast computation and inference speeds.

REFERENCES

- [1] O. Maler and D. Nickovic, "Monitoring Temporal Properties of Continuous Signals," *Lecture Notes in Computer Science*, no. 3253, pp. 152–166, 2004.
- [2] Y. V. Pant, H. Abbas, and R. Mangharam, "Smooth Operator: Control using the Smooth Robustness of Temporal Logic," in *IEEE Conf. on Control Technology and Applications*, 2017.
- [3] Y. Meng and C. Fan, "Diverse Controllable Diffusion Policy With Signal Temporal Logic," *IEEE Robotics and Automation Letters*, vol. 9, no. 10, pp. 8354–8361, 2024.
- [4] W. Liu, N. Mehdipour, and C. Belta, "Recurrent Neural Network Controllers for Signal Temporal Logic Specifications Subject to Safety Constraints," *IEEE Control Systems Letters*, vol. 6, pp. 91 – 96, 2021.
- [5] K. Leung, N. Aréchiga, and M. Pavone, "Backpropagation through signal temporal logic specifications: Infusing logical structure into gradient-based methods," *Int. Journal of Robotics Research*, 2022.
- [6] Z. Zhong, D. Rempe, D. Xu, Y. Chen, S. Veer, T. Che, B. Ray, and M. Pavone, "Guided Conditional Diffusion for Controllable Traffic Simulation," in *Proc. IEEE Conf. on Robotics and Automation*, 2023.
- [7] K. Leung and M. Pavone, "Semi-Supervised Trajectory-Feedback Controller Synthesis for Signal Temporal Logic Specifications," in *American Control Conference*, 2022.
- [8] S. Veer, K. Leung, R. Cosner, Y. Chen, and M. Pavone, "Receding Horizon Planning with Rule Hierarchies for Autonomous Vehicles," in *Proc. IEEE Conf. on Robotics and Automation*, 2023.
- [9] J. DeCastro, K. Leung, N. Aréchiga, and M. Pavone, "Interpretable Policies from Formally-Specified Temporal Properties," in *Proc. IEEE Int. Conf. on Intelligent Transportation Systems*, 2020.
- [10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, 1997.
- [11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is All You Need," in *Conf. on Neural Information Processing Systems*, 2017.
- [12] S. Diamond and S. Boyd, "CVXPY: A Python-Embedded Modeling Language for Convex Optimization," *Journal of Machine Learning Research*, vol. 17, no. 83, pp. 1–5, 2016.
- [13] R. Tedrake *et al.*, "Drake: Model-based design and verification for robotics," Available at <https://drake.mit.edu>, 2019.
- [14] T. Yamaguchi, B. Hoxha, and D. Nickovic, "RTAMT: Runtime Robustness Monitors with Application to CPS and Robotics," *Int. Journal on Software Tools for Technology Transfer*, vol. 26, pp. 79–99, 2023.
- [15] A. Balakrishnan, "Signal Temporal Logic C++ Toolbox," Available at <https://github.com/anand-bala/signal-temporal-logic>, 2021.
- [16] V. Kurtz and H. Lin, "Mixed-Integer Programming for Signal Temporal Logic with Fewer Binary Variables," *IEEE Control Systems Letters*, vol. 6, pp. 2635–2640, 2022.
- [17] G. A. Cardona, K. Leahy, M. Mann, and C.-I. Vasile, "A Flexible and Efficient Temporal LogicTool for Python: PyTeLo," Available at <https://arxiv.org/abs/2310.08714>, 2023.
- [18] M. Vazquez-Chanlatte, "pySTLToolbox," Available at <https://github.com/mvcisback/py-signal-temporal-logic>, 2022.
- [19] E. Bartocci, J. 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," *Lectures on Runtime Verification*, 2018.
- [20] N. Mehdipour, C.-I. Vasile, and C. Belta, "Arithmetic-Geometric Mean Robustness for Control from Signal Temporal Logic Specifications," in *American Control Conference*, 2019.
- [21] J. He, E. Bartocci, D. Ničković, H. Iakovic, and R. Grosu, "DeepSTL – From English Requirements to Signal Temporal Logic," in *IEEE/ACM Int. Conf. on Software Engineering*, 2022.
- [22] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," Available at <http://github.com/google/jax>, 2018.
- [23] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *Conf. on Neural Information Processing Systems - Autodiff Workshop*, 2017.
- [24] J. Song, C. Meng, and S. Ermon, "Denoising Diffusion Implicit Models," in *Int. Conf. on Learning Representations*, 2021.
- [25] M. Janner, Y. Du, J. Tenenbaum, and S. Levine, "Planning with Diffusion for Flexible Behavior Synthesis," in *Int. Conf. on Machine Learning*, 2022.
- [26] K. Mizuta and K. Leung, "CoBL-Diffusion: Diffusion-Based Conditional Robot Planning in Dynamic Environments Using Control Barrier and Lyapunov Functions," in *IEEE/RSS Int. Conf. on Intelligent Robots & Systems*, 2024.
- [27] P. Kapoor, E. Kang, and R. Meira-Góes, "Safe planning through incremental decomposition of signal temporal logic specifications," in *NASA Formal Methods Symposium*. Springer, 2024, pp. 377–396.
- [28] E. Bartocci, N. Manjunath, L. Mariani, C. Mateis, and D. Ničković, "CPSDebug: Automatic failure explanation in CPS models," *Int. Journal on Software Tools for Technology Transfer*, 2021.
- [29] E. Kang, A. Ganlath, S. Mishra, F. Baiduc, and N. Ammar, "Contract-Driven Runtime Adaptation," in *NASA Formal Methods Symposium*, 2024.