Cooperative Sampling-Based Motion Planning under Signal Temporal Logic Specifications

Mayank Sewlia¹, Christos K. Verginis², and Dimos V. Dimarogonas¹

Abstract—We develop a cooperative sampling-based motion planning algorithm for two autonomous agents under coupled tasks expressed as signal temporal logic constraints. The algorithm builds incrementally two spatio-temporal trees, one for each agent, by sampling points in an extended space, which consists of a compact subset of the time domain and the physical space of the agents. The trees are built by checking if newly sampled points form edges in time and space that satisfy certain parts of the coupled task. Therefore, the constructed trees represent time-varying trajectories in the agents’ state space that satisfy the task. The algorithm is distributed in the sense that the agents build their trees individually by communicating with each other. The proposed algorithm inherits the properties of probabilistic completeness and computational efficiency of the original sampling-based procedures.

I. INTRODUCTION

Planning of autonomous agents subject to tasks encoded as temporal logic specifications has attracted a great deal of attention during the last two decades. Expressing tasks as temporal logic constraints offers a great degree of versatility, since such constraints can efficiently describe a large variety of complex planning objectives, as opposed to simple point-to-point navigation. A special form of temporal logic, namely signal temporal logic, offers the incorporation of spatial and time specifications for autonomous agents, providing a rich variety of tasks [1].

There exist numerous works that consider planning under temporal logic specifications. Such specifications include both qualitative properties, such as linear temporal logic (LTL) [2], and quantitative spatio-temporal properties, such as metric temporal logic (MTL) [3]–[6], metric interval temporal logic (MITL) [7]–[10] or signal temporal logic (STL) [11]–[18]. MTL and MITL usually specify tasks over finite-state spaces, requiring the abstraction of the underlying continuous-time and -state systems to discrete ones. However, such abstractions bring several drawbacks, such as loss of information and risk of state explosion. Using STL avoids the aforementioned drawbacks and allows one to express specifications in continuous time and space.

Several works consider the multi-agent motion-planning planning problem under STL tasks [17], [19]–[23]. The work [17], [19] develops distributed feedback control algorithms based on funnel control and barrier functions to guide robots toward satisfying the underlying STL tasks. The limitations of continuous spaces, however, such as the inability to explore the entire time and state spaces, leads to the consideration of only specific STL fragments, losing the full expressivity of STL. Works that consider the entire STL fragment adopt centralized solutions based on mixed-integer linear programming (MILP) [20]–[23]. However, MILP programs can be computationally demanding and yield long running times. Furthermore, centralized algorithms do not scale with the number of agents and are sensitive to faults, since a single computer unit plans the actions of all agents. The general motion-planning problem, even for a two-dimensional configuration space with rectangular obstacles is PSPACE-hard [24], thus, a common approach is sampling-based planners. In such planners, the free space is modeled using a graph with nodes and edges. The motion planning problem then reduces to a graph-search problem where a collision free path is found thereon. By constructing this graph, a multidimensional space search is reduced to finding a path in the graph. This paper develops a distributed algorithm that employs a sampling procedure, inheriting its efficiency and probabilistic completeness properties.

This paper addresses the cooperative motion planning problem of two autonomous agents, which evolve continuously in time and space, to deliver coupled tasks expressed as signal temporal logic specifications. Our contribution with respect to the related literature is the development of a computationally-efficient algorithm that solves, in a distributed manner, the cooperative motion-planning problem under the entire fragment of signal temporal logic for continuous-time and -state systems, without resorting to discretization techniques. Among many places, coupled autonomous-agents find application in warehouses where a load is too heavy for a single manipulator and thus require cooperation between two manipulators to move the load. Multiple-agents can then form teams of coupled-agents to move multiple loads at once.

Inspired by standard sampling-based motion-planning techniques, we develop a cooperative time-augmented sampling-based algorithm. In particular, the algorithm builds incrementally two spatio-temporal trees, one for each agent, by sampling points in the coupled state and time domain of the agents. The algorithm checks if the new edges to be added to the trees satisfy certain parts of the given task. In that way, the resulting spatio-temporal trees represent time-varying trajectories in the agents’ state space that satisfy...
the given spatio-temporal task. Finally, the algorithm can be executed in a distributed manner since the agents build their own trees by communicating with each other. The proposed algorithm is relevant to the approach of [25], which introduces a time-augmented sampling-based algorithm for single-agent systems under STL tasks. Similarly, [18] proposes a sampling-based algorithm for single-agent systems, encoding a temporal-logic task in a cost function to be minimized, without sampling in the time domain.

The rest of the paper is organized as follows. Section II describes the problem considered, Section III provides the proposed algorithm, and Section IV presents simulation results. Finally, Section V concludes the paper.

II. PRELIMINARIES AND PROBLEM FORMULATION

A. Notations

The set of natural numbers is denoted by \( \mathbb{N} \) and the set of real numbers by \( \mathbb{R} \). With \( n \in \mathbb{N} \), \( \mathbb{R}^n \) is the set of \( n \)-coordinate real-valued vectors and \( \mathbb{R}^n_+ \) is the set of real \( n \)-vector with non-negative elements. The cardinality of a set \( A \) is denoted by \( |A| \). If \( a \in \mathbb{R} \) and \( [b, c] \in \mathbb{R}^2 \), the Kronecker sum \( a \oplus [b, c] = [a + b, a + c] \in \mathbb{R}^2 \). We further define the boolean set as \( \mathbb{B} = \{ \top, \bot \} \) (True, False).

B. Signal Temporal Logic

Let \( x : \mathbb{R}_{\geq 0} \to \mathbb{R}^n \) be a continuous-time signal. Signal temporal logic [26] is a predicate-based logic with the following syntax:

\[
\varphi = \top \mid \mu \mid \neg \varphi \mid G_{[a,b]} \varphi \mid F_{[a,b]} \varphi \mid \varphi_1 U_{[a,b]} \varphi_2 \mid \varphi_1 \land \varphi_2
\]

(1)

where \( \varphi_1, \varphi_2 \) are STL formulas and \( U_{[a,b]} \) encodes the until operator, with \( 0 \leq a \leq b < \infty \); \( \mu \) is a predicate of the form \( \mu : \mathbb{R}^n \times \mathbb{R}_{\geq 0} \to \mathbb{B} \) defined via a predicate function \( h : \mathbb{R}^n \times \mathbb{R}_{\geq 0} \to \mathbb{R} \) as

\[
\mu = \begin{cases} 
\top & h(x,t) \geq 0 \\
\bot & h(x,t) < 0 
\end{cases} .
\]

(2)

We consider time bounded temporal operators. The satisfaction relation \( (x,t) \models \varphi \) indicates that signal \( x \) satisfies \( \varphi \) at time \( t \) and is defined recursively as follows:

\[
\begin{align*}
(x,t) \models \mu & \iff h(x,t) \geq 0 \\
(x,t) \models \neg \varphi & \iff \neg(h(x,t) \mid \varphi) \\
(x,t) \models \varphi_1 \land \varphi_2 & \iff (x,t) \models \varphi_1 \land (x,t) \models \varphi_2 \\
(x,t) \models \varphi_1 U_{[a,b]} \varphi_2 & \iff \exists t_1 \in [t + a, t + b] \text{ s.t. } (x,t_1) \models \varphi_2 \\
& \land \forall t_2 \in [t, t_1], (x,t_2) \models \varphi_1.
\end{align*}
\]

We also define the operators *disjunction*, *eventually*, and *always* as \( \varphi_1 \lor \varphi_2 \equiv \neg (\neg \varphi_1 \land \neg \varphi_2) \), \( F_{[a,b]} \varphi \equiv \top U_{[a,b]} \varphi \), and \( G_{[a,b]} \varphi \equiv \neg F_{[a,b]} \neg \varphi \), respectively. The satisfaction relation \( (x,t) \models \varphi \) can also be quantified as follows:

\[
\begin{align*}
\rho(h(x,t) \geq 0, x,t) &= h(x(t),t) \\
\rho(\neg \varphi, x,t) &= -\rho(\varphi, x,t) \\
\rho(\varphi_1 \land \varphi_2, x,t) &= \min(\rho(\varphi_1, x,t), \rho(\varphi_2, x,t)) \\
\rho(G_{[1,\tau]} \varphi, x,t) &= \inf_{t_1 \in t \tau} \rho(\varphi, x,t_1) \\
\rho(F_{[\rho]} \varphi, x,t) &= \sup_{t_1 \in t \rho} \rho(\varphi, x,t_1) \\
\rho(\varphi_1 U_{[\rho]} \varphi_2, x,t) &= \sup_{t_1 < t, t_2 \in [t,t_1]} \min(\rho(\varphi_2, x,t_1), -\inf_{t_2 \in [t,t_1]} \rho(\varphi_1, x,t_2)).
\end{align*}
\]

A signal \( x \) satisfies an STL formula \( \varphi \) at time \( t \) if and only if \( \rho(\varphi, x,t) \geq 0 \).

C. STL Parse Tree

An STL formula defined recursively over the form (1) can be represented as a tree, we name it as an STL parse tree. An STL parse tree is constructed as follows:

- each node is either a temporal operator \( \{G_{[1,\tau]} \}, \{F_{[\rho]} \}, \{\land \}, \{\lor \}, \{\neg \} \) or a predicate \( \{\mu\} \) where \( I \subset \mathbb{R} \) is a closed interval,
- each node, apart from predicate nodes, is accompanied by a satisfaction variable \( \tau \in \{+1, -1\} \); such nodes are termed set nodes,
- a root node has no parent node and a leaf node has no child note. The leaf nodes constitute the predicate nodes of the tree.

A path is a formula from a root node to a leaf node and we see that set of all paths constitutes the tree. A sub-path (or a sub-formula) is a path from a set node to a leaf node. Each set node is accompanied by a satisfaction variable \( \tau \in \{+1, -1\} \) and each leaf node is accompanied by a predicate variable \( \pi = \mu \). A signal \( x \) satisfies a sub-path if \( \tau = +1 \) corresponding to the set node where the path begins. An analogous tree of satisfaction and predicate variables can be drawn, called satisfaction variable tree. The satisfaction variable tree borrows the same tree structure as the STL parse tree. Each set node from the STL parse tree maps uniquely to a satisfaction variable \( \tau_i \) and each leaf node maps uniquely to a predicate variable \( \pi_i \), where \( i \) is an enumeration of the nodes in the satisfaction variable tree.

The STL parse tree and the satisfaction variable tree for the STL formula

\[
\varphi = F_{[\tau_1]} (\mu_1 \lor G_{[\tau_2]} (\mu_2)) \land G_{[\tau_3]} F_{[\tau_4]} \mu_3 \land G_{[\tau_5]} \mu_4.
\]

are shown in Figure 1. Suppose \( \tau_2 = +1 \implies x \models F_{[\tau_1]} (\mu_1 \lor G_{[\tau_2]} (\mu_2)) \). And if \( \tau_1 = +1 \implies x \models G_{[\tau_3]} \mu_3 \). We then have the following equivalence,

\[
(x,t) \models \varphi \iff \rho(\varphi, x,t) \geq 0 \iff \tau(\text{root}) = +1.
\]

Before we proceed to the problem formulation, we need the following definition:
Fig. 1: STL parse tree and satisfaction variable tree for the formula in (3).

**Definition 1 ([27]):** The time horizon ‘th(φ)’ of an STL formula φ is recursively defined as,

\[
\text{th}(\varphi) = \begin{cases} 
0, & \text{if } \varphi = \mu \\
\text{th}(\varphi_1), & \text{if } \varphi = \neg \varphi_1 \\
\max\{\text{th}(\varphi_1), \text{th}(\varphi_2)\}, & \text{if } \varphi = \varphi_1 \land \varphi_2 \\
b + \max\{\text{th}(\varphi_1), \text{th}(\varphi_2)\}, & \text{if } \varphi = \exists_{[a,b]} \varphi_2.
\end{cases}
\]  

(5)

**D. Problem Formulation**

In this paper, we consider the motion-planning problem of two autonomous agents subject to a coupled task expressed as an STL formula φ of the form (1). We assume that the agents evolve in \(\mathbb{R}^n\), with \(n \in \mathbb{N}\), characterized by the state trajectories \(x^i : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^n\), for \(i \in \{1, 2\}\). More specifically, the problem we consider is the design of time-varying trajectories \(y_i : [0, \infty) \rightarrow \mathbb{R}^n\), starting at the initial configurations \(y_i(0) = x^i(0), i \in \{1, 2\}\), that satisfy a user-defined coupled STL task φ, i.e., \((y, 0) \models \phi\), with \(y := [y_1^T, y_2^T]^T\).

We currently do not focus on dynamics of the agent or the explicit control design that guarantees tracking of the trajectory \(y_i(t)\); we assume that the agents are endowed with a control algorithm that accomplishes such tracking. Instead, we focus on the planning problem, i.e., the derivation of continuous time-varying trajectories that satisfy φ. Unlike previous works in the related literature (e.g., [16]), we consider the entire fragment of STL and develop a distributed and efficient sampling-based algorithm.

**III. Sampling-Based Planning Under STL**

This section provides the proposed solution to the cooperative STL planning problem. The solution consists of a variation of the standard Rapidly-exploring Random Tree (RRT) algorithm [28] and it derives a time-varying trajectory \(y(t)\) that satisfies φ. The key element is the incorporation of time in the sampling process, leading to a time-augmented sampling method. Compared to the original sampling-based algorithms, we replace the obstacle collision-checking procedure with a procedure that checks whether a specific fragment of the constructed tree violates the given STL formula. The aforementioned properties allow the direct derivation of a time-varying trajectory \(y(t)\) that satisfies the given STL task encoded in φ. This problem poses additional challenges since it involves determining satisfaction of φ, which is defined over a continuous signal, using only point-wise data. We propose the following algorithm, called STLcoRRT, to address these challenges.

The STLcoRRT algorithm comprises two main components, namely the sampling-based procedure to build a tree, and the STL verification procedure that decides the satisfaction of a STL formula. Each sample corresponds to a point \(z = (t, x) \in Z \subset \mathbb{R}_+ \times \mathbb{R}^n\), which the STL verification procedure decides if it is feasible with respect to the STL formula φ; if yes, it approves it and adds it as a vertex to the tree. Every added edge is verified against a formula and a path traversing any edge ensures the satisfaction of φ by construction. The termination of the STLcoRRT algorithm guarantees the generation of a trajectory that satisfies the formula.

In what follows, we introduce first the sampling-based procedure in Section III-A and then the overall STLcoRRT algorithm in Section III-B.

**A. Sampling-based Procedure**

The traditional RRT algorithm samples a point \(x_{\text{samp}}\) from \(\mathbb{R}^n\), finds its nearest neighbour \(x_{\text{nearest}}\) from the existing tree \(T = (V, E)\), and draws an edge of predefined length step to create a new vertex \(x_{\text{new}}\). The inputs to the RRT algorithm are a starting vertex \(x(0)\), representing the initial condition of a path, a goal region, signifying the termination of the algorithm (once \(x_{\text{new}}\) is sampled in the goal region), and a hyper-parameter step representing the edge length. An edge from \(x_{\text{nearest}}\) to \(x_{\text{new}}\) is added to \(E\) in the direction of \(x_{\text{samp}}\). The Euclidean distance between \(x_{\text{new}}\) and \(x_{\text{nearest}}\) is user defined and denoted by \(x_{\text{step}}\). The STLcoRRT algorithm, as opposed to the traditional RRT algorithm, generates a spatio-temporal tree \(T = (V, E)\), where \(V\) is the set of vertices and \(E\) is the set of edges. Each vertex in the set \(V = \{z_1, z_2, \ldots\}\) is given by \(z_i = (t_i, x_i) \in Z \subset \mathbb{R}_+ \times \mathbb{R}^n\). By sampling in time, the termination of STLcoRRT generates a trajectory in space and time while RRT only returns a path in space. The continuous sample space \(Z\) is predefined based on the range of variables \(x_i\) and the time horizon of the STL formula as defined in Definition 1. We assume the states \(x_i\) are sampled from a closed set. Another distinction is that in STLcoRRT, a vertex \(z_{\text{new}} = (t_{\text{new}}, x_{\text{new}})\) is sampled if and only if \(t_{\text{new}} > t_{\text{nearest}}\) and subsequently any \(z_{\text{new}}\) mentioned henceforth is considered to be ahead in time with respect to \(z_{\text{nearest}}\). Along with this restriction, every eligible \(z_{\text{new}}\) goes through a STL verification process deciding if the vertex could be a satisfactory point in the final trajectory satisfying \((y(t) \models \phi)\).

**B. Overall Algorithm**

To solve the coupled motion planning problem for two cooperating agents, agent \(i\) and agent \(j\), we partition the algorithm into three modules. From the point of view of agent \(i\), the first module is about sampling in a time region where agent \(j\) has already explored, i.e. Algorithm 3: PastAgent. The second module is sampling in a time region where agent
Algorithm 1: STLcoRRT: Agent $i$ perspective

**Input:** $\varphi$, $z_i^0$, $\text{step}$, $L$, $Z^i_{\text{goal}}$, and $Z$  
**Output:** $T_i$ and $\text{path}_i$  

```
1  $\mathcal{V}_i \leftarrow \{z_i^0\}$; 
2  $\mathcal{E}_i \leftarrow \emptyset$; 
3  $T_i \leftarrow (\mathcal{V}_i, \mathcal{E}_i)$; 
4  $t_0 \leftarrow 0$; 
5  $\text{path}_i \leftarrow \emptyset$; 
6  $\tau \leftarrow \{-1\}$; 
7  while $z_i^0 \notin Z^i_{\text{goal}}$ do 
8      $T_j \leftarrow \text{Agent } j$; 
9      while $z_{\text{new},j}^i < z_{\text{nearest},j}^i$ do 
10         $z_{\text{samp}} \leftarrow \text{Sample}(Z)$; 
11         $z_{\text{nearest}} \leftarrow \text{Nearest}(T_i, z_{\text{samp}})$; 
12         $z_{\text{new}} \leftarrow \text{Steer}(z_{\text{nearest}}, T_i, \text{step})$; 
13         if $z_{\text{new}} > \max \{\mathcal{V}_j, t\}$ then 
14            $T_i, T_j \leftarrow \text{FutureAgent}(\varphi, \text{step}, L, z_{\text{new}}^i, z_{\text{nearest}}, \tau, T_i, T_j)$; 
15         else 
16            $T_i \leftarrow \text{PastAgent}(\varphi, L, z_{\text{new}}^i, z_{\text{nearest}}, \tau, T_i, T_j)$; 
17      $\text{path}_i \leftarrow \text{path}_i \cup \text{PlotPath}(T_i)$; 
```

Algorithm 2: FutureAgent: Agent $i$ perspective  

```
1  $z_{\text{new}}^i = \text{Sample}(Z)$; 
2  $z_{\text{nearest}}^i = \text{Nearest}(T_i, z_{\text{samp}})$; 
3  $z_{\text{new}}^i = \text{Steer}(z_{\text{nearest}}, \tau, T_i, T_j)$; 
4  $\text{path}_i \leftarrow \text{path}_i \cup \text{PlotPath}(T_i)$; 
```

Algorithm 3: PastAgent: Agent $i$ perspective  

```
1  $z_{\text{new}}^i = \text{Sample}(Z)$; 
2  $z_{\text{nearest}}^i = \text{Nearest}(T_i, z_{\text{samp}})$; 
3  $z_{\text{new}}^i = \text{Steer}(z_{\text{nearest}}, \tau, T_i, T_j)$; 
4  $\text{path}_i \leftarrow \text{path}_i \cup \text{PlotPath}(T_i)$; 
```

Algorithm 4: PlotPath: Agent $i$ perspective  

```
1  $\text{path}_i \leftarrow \text{path}_i \cup \text{PlotPath}(T_i)$; 
```

j is yet to explore, i.e. Algorithm 2: FutureAgent. And the final module is verification of the STL formula based on the sampled points, i.e. Algorithm 4: EdgeApproval. For the discussions below, we use index $i$ to refer to an agent that is currently running the pseudo-code and index $j$ for the other agent. We now describe the algorithm and modules in detail.

1) Algorithm 1 - STLcoRRT: The main body of STLcoRRT starts with the initialization of the tree $T_i$ (not to be confused with the STL parse tree and the satisfaction variable tree) with the initial conditions $z_i^0 = (0, x_i(0))$ (line 1-5). In the pseudo-code, we denote the time component of a sample $z^i = (t^i, x^i)$ as $z^i = z^i_{\text{new}}$ i.e., $t^i := z^i_{\text{new}}$. Agent $i$ then requests the updated tree of agent $j$ (line 8). The sampling takes place between lines 10-12 where the functions Sample, Nearest and Steer are standard and borrowed from the original RRT algorithm [28]:

- Sample samples a random point $z_{\text{samp}}^i = (t_{\text{samp}}, x_{\text{samp}})$ in $Z$.
- Nearest finds the nearest node $z_{\text{nearest}}^i = (t_{\text{nearest}}^i, x_{\text{nearest}}^i)$ with respect to the Euclidean distance in $T_i$ to the sampled point $z_{\text{samp}}$ and.
- Steer finds a node $z_{\text{new}}^i = (t_{\text{new}}, x_{\text{new}})$ which is step units apart from $z_{\text{nearest}}^i$ in the direction of $x_{\text{samp}}$.

The condition in line 9 imposes every sample to be sampled ahead in time compared to its nearest neighbour. This guarantees every path in the tree to only move forward in time. Each agent samples points sequentially and builds its tree after communicating with the other agent. We distinguish two scenarios in this sampling procedure: a new sample $z_{\text{new}}^i = (t_{\text{new}}, x_{\text{new}})$ of one agent is at a point in time $t_{\text{new}}$ when (1) the other agent $j$ hasn’t built a tree yet i.e. $t_{\text{new}}>\max\{t_k^j\}$, $k \in \{1, \ldots, |V_j|\}$ or (2) the other agent $j$ has existing edges passing through this time instant $t_{\text{new}}$ i.e $t_{\text{new}} < \max\{t_k^j\}$, $k \in \{1, \ldots, |V_j|\}$. In the former case, each agent uses the function FutureAgent and in the latter case each agent uses the function PastAgent to decide the satisfiability of $\varphi$. This process is repeated until $z_i^i \in Z^i_{\text{goal}}$ and terminates by finding the trajectory $\text{path}_i$. The goal region is $Z^i_{\text{goal}} = \{z_i^i \in Z \mid t_i^i \geq \theta_i(\varphi)\}$.

2) Function 2 - FutureAgent: This function is called when agent $i$ samples a point $z_{\text{new}}^i = (t_{\text{new}}, x_{\text{new}})$ where $t_{\text{new}} > t_k^j$ for all $k \in \{1, 2, \ldots, |V_j|\}$ as seen in Figure 2a (left). It starts by temporarily adding the sampled vertex $z_{\text{new}}$ to the tree of agent $i$ (lines 1-2). Next, it communicates this $z_{\text{new}}$ to agent $j$ and agent $j$ samples indefinitely until a sampled point lies incident to the temporarily added edge of agent $i$, lines 3-6 and seen in Figure 2a (center). Then, the algorithm linearly interpolates in line 7 to obtain the vertex $z_{\text{inter}} = (t_{\text{new}}, x_{\text{inter}})$ as seen in Figure 2a (right). This is obtained by solving for $x_{\text{inter}}$ element-wise as the solution of,

$$
x_{\text{inter}} = \left( x_{\text{nearest}}^i - x_{\text{new}}^i \right) \frac{t_{\text{inter}} - t_{\text{new}}}{t_{\text{nearest}}^i - t_{\text{new}}} + x_{\text{new}}^i.
$$

We define a function Interpolate that solves the aforementioned computation. We can now sample points over the edge $\{z_{\text{nearest}}^i, z_{\text{inter}}^i\}$ and all the corresponding interpolated points from $T_j$ to evaluate the STL formula through the EdgeApproval function, depicted in Function 4 and discussed below. By $L \in \mathbb{N}$ (see Section III-B.1) we chose the
number of equally spaced points on the edge \( \{z_i^{\text{nearest}}, z_i^{\text{new}}\} \) in the evaluation of \( \varphi \). The new vertex \( z_i^{\text{new}} \) of agent \( i \) is deleted if it does not satisfy the STL formula (lines 13-14).

**Function 2: FutureAgent**

Input: \( \varphi, \text{step}, L, z_i^{\text{new}}, z_i^{\text{nearest}}, \tau, T_i, \) and \( T_j \)

Output: \( T_{i} \) and \( T_{j} \)

1. \( V_i \leftarrow V_i \cup \{z_i^{\text{new}}\} \)
2. \( E_i \leftarrow E_i \cup \{z_i^{\text{new}}, z_i^{\text{nearest}}\} \)
3. while \( z_i^{\text{new}}, t < z_i^{\text{nearest}}, t \) or \( z_i^{\text{new}}, t \notin (z_i^{\text{nearest}}, t, z_i^{\text{new}}, t) \) do
   4. \( z_i^{\text{samp}} \leftarrow \text{Sample}(Z) \)
   5. \( z_i^{\text{nearest}} \leftarrow \text{Nearest}(T_i, z_i^{\text{samp}}) \)
   6. \( z_i^{\text{new}} \leftarrow \text{Steer}(z_i^{\text{nearest}}, z_i^{\text{new}}, \text{step}) \)
   7. \( z_i^{\text{inter}} \leftarrow \text{Interpolate}(\{z_i^{\text{nearest}}, z_i^{\text{new}}\}, z_i^{\text{new}}, t) \)
   8. for \( l \leftarrow 0 \) to \( L \) by \( 1/(L-1) \) do
      9. \( z_i^{l} = z_i^{\text{nearest}} + l(z_i^{\text{new}} - z_i^{\text{nearest}}) \)
     10. \( z_i^{l} \leftarrow \text{Interpolate}(\{z_i^{\text{nearest}}, z_i^{\text{new}}\}, z_i^{\text{l}}, t) \)
     11. \( \pi, \tau \leftarrow \text{EdgeApproval}(\varphi, z_i^{l}, z_i^{\text{l}}, \tau) \)
     12. if \( \pi \) then
        13. \( V_i \leftarrow V_i \cup \{z_i^{\text{l}}\} \)
        14. \( E_i \leftarrow E_i \cup \{z_i^{\text{new}}, z_i^{\text{l}}\} \)
        15. \text{return} \( T_{i} \) and \( T_{j} \)

16. \( V_i \leftarrow V_i \cup \{z_i^{\text{new}}\} \)
17. \( E_i \leftarrow E_i \cup \{z_i^{\text{nearest}}, z_i^{\text{new}}\} \)
18. \( V_i \leftarrow V_i \cup z_i^{\text{new}} \)
19. \( E_i \leftarrow E_i \cup \{z_i^{\text{new}}, z_i^{\text{new}}\} \)
20. \( V_i \leftarrow V_i \cup \{z_i^{\text{new}}, z_i^{\text{new}}\} \)
21. \( E_i \leftarrow E_i \cup \{z_i^{\text{new}}, z_i^{\text{new}}\} \)

**Function 3: PastAgent**

Input: \( \varphi, L, z_i^{\text{new}}, z_i^{\text{nearest}}, \tau, T_i, \) and \( T_j \)

Output: \( T_{i} \)

1. foreach \( \{z_i^{m}, z_i^{m+1}\} \in E_j \) do
   2. \( t_i^{m} \leftarrow z_i^{m}, t_i^{m} \)
   3. \( t_i^{m+1} \leftarrow z_i^{m+1}, t_i^{m+1} \)
   4. if \( t_i^{m} < t_i^{\text{new}} < t_i^{m+1} \) then
      5. for \( l \leftarrow 0 \) to \( L \) by \( 1/(L-1) \) do
         6. \( z_i^{l} = z_i^{m+1} + l(z_i^{\text{new}} - z_i^{m}) \)
         7. \( z_i^{l} \leftarrow \text{Interpolate}(\{z_i^{m}, z_i^{m+1}\}, z_i^{l}, t) \)
         8. \( \pi, \tau \leftarrow \text{EdgeApproval}(\varphi, z_i^{l}, z_i^{\text{l}}, \tau) \)
         9. if \( \pi \) then return \( T_{i} \)
      10. \( V_i \leftarrow V_i \cup \{z_i^{\text{l}}\} \)
      11. \( E_i \leftarrow E_i \cup \{z_i^{\text{new}}, z_i^{\text{l}}\} \)

4) Encoding STL formula into the algorithm via Function 4 (EdgeApproval): Function 4 is recursively defined and takes as input an STL formula of the form (1), a pair of points \( \{z^i, z^j\} \) from the trees of both agents, and returns a truth value \( \pi \) to determine if the vertices satisfy the STL formula. Along with \( \pi \), it returns a variable \( \tau(\varphi) \) taking values in the set \{1, -1\} indicating satisfaction of the STL sub-formula \( \varphi \). The satisfiability of an STL formula cannot be determined by an individual point \( z_i^{\text{new}} \) along with \( z_i^{\text{new}} \). However, a point \( z_i^{\text{new}} \) along with \( z_i^{\text{new}} \) can interconnectively determine the satisfaction at that particular instant in time, since the formula consists of both temporal and spatial variables. Next, we present a set of rules for adding a pair of sampled points \( z_i^{\text{new}}, z_i^{\text{new}} \) to the tree based on the STL formula \( \varphi \) in (1).

- \( \varphi = T \): Any sampled point \( z_i^{\text{new}} \) is added to the tree.
- \( \varphi = \mu \): A sampled point \( z_i^{\text{new}} \) is added to the tree, if and only if \( \mu = T \) with
  \[ 
  \mu = \begin{cases} 
  T & \text{if } h(x_i^{\text{new}}, x_i^{\text{new}}, t_i^{\text{new}}) \geq 0 \\
  \bot & \text{if } h(x_i^{\text{new}}, x_i^{\text{new}}, t_i^{\text{new}}) < 0
  \end{cases}
  \]
And it is also possible that such a point might not be a part of the final trajectory. Thus, to ensure this satisfying point is included in our final trajectory we first extract and store the path connecting \( z^t_i \) and \( z^t_{\text{new}} \). The trees of both agents are then reset and only the nodes \( z^t_{\text{new}} \) and \( z^t_{\text{new}} \) are added to \( V_i \) and \( V_j \) respectively. By doing this, any final trajectory will pass through \( z^t_{\text{new}} \) as it is the only point connecting the prior tree and the new tree, lines 21-26. For \( I = [a, b] \), if \( z^t_{\text{new}} < b \) the sampled point \( x^t_{\text{new}} \) is added to the tree. Any sampled point \( x^t_{\text{new}} > b \) is not added to the tree, this stops the tree from growing beyond \( x^t_{\text{new}} > b \) until we find a satisfying point. Once a point is sampled that satisfies \( \varphi \), we remove the hold on tree expansion i.e. the tree is allowed to expand beyond \( x^t_{\text{new}} > b \). Once \( \tau(\varphi) = +1 \), then this implies that \( \tau(\mathcal{F}_1 \varphi) = +1 \) for the sample \( z^t_{\text{new}} \). Note that if \( \varphi = \mu \), then \( \tau(\varphi) = +1 \Rightarrow \mu = T \).

\[ \varphi = \mathcal{G}_1 \varphi \]: If \( z^t_{\text{new}} \in [a, b] \), the sampled point \( z^t_{\text{new}} \) is added to the tree if \( \tau(\varphi) = +1 \). If \( t_{\text{new}} \notin [a, b] \) then \( z^t_{\text{new}} \) is added to the tree. This means the algorithm does not proceed if it does not satisfy \( \mathcal{G}_1 \varphi \). Again note \( \tau(\varphi) = +1 \Leftrightarrow \mu = T \) if \( \varphi = \mu \).

\[ \varphi = \varphi_1 \cup \varphi_2 \]: The until operator is treated as \( \mathcal{G}[z^t_{\text{new}}, t_{\text{new}}] \varphi_1 \) until \( \tau(\mathcal{F}[z^t_{\text{new}}, t_{\text{new}}] \varphi_1) = +1 \). The interval is a singleton set since the satisfaction instance of \( \varphi_2 \) cannot be assessed apriori.

**Function 4: EdgeApproval**

**Input:** \( \varphi, z^t_{\text{new}}, F^t_{\text{new}}, \tau \)

**Output:** \( \pi, \tau \)

```plaintext
1 switch \( \varphi \) do
2   case \( \top \) do
3     return \( \pi, \tau \)
4     case \( \mu \) do
5       if \( \mu \) then return \( \pi, \tau \);
6       else return \( \bot, \tau \);
7     case \( \neg \varphi \) do
8       return \( \neg \text{EA}(\varphi, \tau) \)
9     case \( \varphi_1 \lor \varphi_2 \) do
10    if \( z^t_{\text{new}} \in t_0 \cup I \) and \( \text{EA}(\varphi_1 > \tau) = (\pi, +1) \) then
11      return \( \text{EA}(\varphi_1, \tau) \lor \text{EA}(\bot, \tau) \);
12    else if \( z^t_{\text{new}} \in t_0 \cup I \) and \( \text{EA}(\varphi_2 > \tau) = (\pi, +1) \) then
13      return \( \text{EA}(\bot, \tau) \lor \text{EA}(\varphi_2, \tau) \);
14    else
15      return \( \neg \text{EA}(\varphi_1, \tau) \lor \text{EA}(\varphi_2, \tau) \);
16    case \( \mathcal{F}_1 \varphi \) do
17      if \( \mathcal{G}_{\text{new}} \in t_0 \cup I \) and \( \text{EA}(\varphi, \tau) = (\pi, +1) \) then
18        return \( \text{EA}(\mathcal{F}_1 \varphi, \tau) = (\pi, +1) \); pathi1 <- PlotPath(\( \mathcal{F}_1 \));
20        pathi2 <- PlotPath(\( \mathcal{F}_2 \));
22        if \( \mathcal{G}_{\text{new}} \in t_0 \cup I \) then
24          return \( \text{EA}(\mathcal{F}_1 \varphi, \tau) = (\pi, +1) \);
26      else
27        return \( \text{EA}(\mathcal{F}_1 \varphi, \tau) = (\pi, +1) \);
29      return \( \neg \text{EA}(\varphi, \tau) \);
30      else
31        return \( \text{EA}(\mathcal{F}_1 \varphi, \tau) = (\pi, +1) \);
33      return \( \neg \text{EA}(\mathcal{F}_1 \varphi, \tau) \);
35      if \( z^t_{\text{new}} \in t_0 \cup I \) then
37      return \( \text{EA}(\varphi, \tau) \);
39      else
41      return \( \neg \text{EA}(\varphi, \tau) \);
43      if \( z^t_{\text{new}} > I \) then \( \tau = +1 \); \( \tau = +1 \); \( \tau = +1 \); \( \tau = +1 \);
45      return \( \text{EA}(\mathcal{F}_1 \varphi, \tau) = (\pi, +1) \);
47      return \( \text{EA}(\mathcal{F}_1 \varphi, \tau) = (\pi, +1) \);
49      return \( \neg \text{EA}(\mathcal{F}_1 \varphi, \tau) = (\pi, +1) \);
51      return \( \neg \text{EA}(\mathcal{F}_1 \varphi, \tau) = (\pi, +1) \);
/* EdgeApproval is abbreviated as EA for readability purposes */
```

The function \( \text{EdgeApproval} \) is recursively defined over (1) and hence covers the entire STL formula. Below we see an example of how \( \text{EdgeApproval} \) treats nested temporal operators.

**Example 1:** Let \( \varphi = G_{[0,2]} F_{[1,3]} \mu \). Formula \( \varphi \) requires \( \mu \) to hold at least every two seconds in the interval [1, 23].
Recalling $\text{EdgeApproval}(\varphi)$, once $\mu = T$ in the interval $[1, 3]$, line 20 in Function 4 returns $\tau(F_{[1,3]}\mu) = +1$ and the initial conditions are set to $x_0 \leftarrow z_{\text{new}}$. This ensures $t_0 \rightarrow z_{\text{new}}$. The next case is $G \varphi$ which returns $G(T)$ in line 34 and $\text{EdgeApproval}$ returns $T, +1$. This process repeats until the entire interval $[0, 23]$ is covered.

Once the algorithm terminates i.e. when the tree reaches the designated goal region, a path $(y_1(t), y_2(t))$ is generated by PlotPath as done in the standard RRT [28]. The paths $(y_1(t), y_2(t))$ satisfy the STL formula by construction since every node in the tree satisfies the STL formula via the $\text{EdgeApproval}$ function. The generated trajectories can be verified using the robust semantics presented in Section II-B by evaluating for $\rho (\varphi, x, t)$. STLcoRRT, unlike monitoring algorithms, generates trajectories that satisfy the STL formula by construction. The inter-dependency of coupled tasks between agents poses a significant challenge in solving the problem in a distributed way while covering the entire STL between agents poses a significant challenge in solving the problem for all agents.

The time complexity, Completeness and Scalability: The time complexity is the sum of individual time complexities of the functions in STLcoRRT, i.e. $C_{\text{STLcoRRT}}(N) = C_{\text{Sample}}(N) + C_{\text{Nearest}}(N) + C_{\text{Steer}}(N) + C_{\text{FutureAgent}}(N) + C_{\text{PastAgent}}(N)$. The time complexity of $C_{\text{Sample}}(N) + C_{\text{Nearest}}(N) + C_{\text{Steer}}(N)$ is $O(N) + O(N) + O(N) = O(N)$. Each sampled vertex belongs to either FutureAgent or PastAgent. Thus the time complexity of FutureAgent and PastAgent is the worst case time complexity of either one of them.

The time complexity of FutureAgent is $C_{\text{FutureAgent}}(N) = C_{\text{Interpolate}}(N) + C_{\text{Interpolate}}(LN) + C_{\text{EdgeApproval}}(N) + O(N + O(LN) + O(N \cdot \log(N))) = O(N \cdot \log(N))$. Similarly $C_{\text{PastAgent}}(N) = O(N \cdot \log(N))$. Thus the worst case time complexity of FutureAgent and PastAgent is still $O(N \cdot \log(N))$. Finally, the time complexity of $C_{\text{STLcoRRT}}(N) = O(N \cdot \log(N))$. The space complexity for individual agents is the size of the stored tree and is thus $O(N)$ where $N$ is the number of vertices. The STLcoRRT algorithm inherits the probabilistic completeness property from the standard sampling algorithm RRT. The interpolation of temporarily added edges in FutureAgent ensures such completeness since non-feasible vertices aren’t added to the tree. As with sampling based methods, we can take various steps to speed up the algorithm such as:

- Changing the step to a larger value for high dimensions;
- A maximum number of nodes can be specified in Algorithm 1 by changing Line 7 to a for loop;
- An iteration limit can be imposed on the second condition in Line 3 of Function 2. Since in some cases it could be too narrow to sample in $z_{\text{nearest}}, z_{\text{new}}$.

IV. SIMULATIONS

This section presents some simulation examples demonstrating the functioning of the STLcoRRT algorithm. We start with simulating examples from the fragment $\varphi = [G_{[0,6]}(x_1 - x_2)]$, where $\varphi_2$. All simulations are performed on an eight core 1.9 GHz Intel-core i7 CPU with 16GB of RAM. Figure 3a simulates the case when $\varphi = [x_1 - x_2 > 5$, requiring the agents to be 5 units apart at all times. Agent 1 (in blue) and Agent 2 (in green) build their trees ensuring that every added vertex is at least 5 units away from all incident vertices of the other agent such that any final trajectory will satisfy $\varphi$. Figure 3b simulates the formula $\varphi = G_{[4,6]}(x_1 - x_2) < 2$ requiring in the interval $[4, 6]$ units, every added vertex is less than two units apart with respect to the other agent. The trees are built satisfying this constraint in the interval $[4, 6]$. In Figure 3c, we simulate the formula $\varphi = F_{[0,6]}(x_1 - x_2) > 8$ requiring the agents to be 8 units apart at any one instant in the interval $[4, 6]$. All sampled vertices $z_{\text{new}}$ are added to the tree until $|x_1 - x_2| > 8$ becomes true, indicated by red crosses in Figure 3c. Once the condition $|x_1 - x_2| > 8$ holds, the corresponding vertices are set as new starting vertices for the STLcoRRT algorithm and are thus included in the final path. Figure 3d requires $x < 1$ in the interval $[2, 8]$ and $|x_1 - x_2| < 2$ in the interval $[4, 6]$. This is evident from the figure as Agent 1 samples only in the set $x < 1$ in the interval $[2, 8]$ and Agent 2 comes closer satisfying $|x_1 - x_2| < 2$ in the interval $[4, 6]$. Next, we present a disjunction example, let $\varphi = F_{[4,6]}(x_1 - x_2) > 8 \lor G_{[6,8]}(x_1 - x_2) < 2$. In Figure 4a (Left), we see that Algorithm 1 chose to satisfy the subformula $F_{[4,6]}(x_1 - x_2) > 8$ while in other run $G_{[6,8]}(x_1 - x_2) < 2$ was satisfied as seen in Figure 4a (Right). For the case of a nested formula, $\varphi = G_{[0,10]}(F_{[1,3]}(x_1 - x_2) > 8$, which dictates a task to ‘satisfy $|x_1 - x_2| > 8$ every 2 seconds in the interval $[1, 13]$’. As seen in Figure 4b, each red cross represents the satisfaction of $F_{[x_1, x_2]}$ where $[x^1, x^2]$ are set to $[1, 3]$ at $t = 0$ and re-evaluated after every satisfaction of the eventually.

V. CONCLUSION

This paper presents an efficient sampling-based algorithm for cooperative motion planning between two agents under STL specifications. We augment the traditional RRT method to generate spatio-temporal trees through which we enforce space and time constraints specified by the STL. The proposed algorithm is distributed as agents independently build their trees by communicating with each other. For future work, we will consider robust STL symantics and extend the existing framework to multiple cooperating agents.

REFERENCES

Fig. 3: Trajectories generated by Algorithm 1 for a single predicate, always, eventually and conjunction operators.

(b) $\phi = |x_1 - x_2| > 5$

(c) $\phi = F_{[4,6]}(|x_1 - x_2| > 8)$

(d) $\phi = G_{[2,6]}(|x_1 - x_2| < 2)$

Fig. 4: Trajectories generated by Algorithm 1 for disjunction operator and nested formula.

(a) $\phi = |x_1 - x_2| > 5$

(b) $\phi = G_{[4,6]}(|x_1 - x_2| < 2)$

(c) $\phi = F_{[4,6]}(|x_1 - x_2| > 8)$

(d) $\phi = G_{[2,6]}(|x_1 - x_2| < 2)$

(a) $\phi = F_{[4,6]}|x_1 - x_2| > 8 \lor G_{[6,8]}|x_1 - x_2| < 2$

(b) $\phi = G_{[0,10]}F_{[1,3]}|x_1 - x_2| > 8$


