2019 IEEE 58th Conference on Decision and Control (CDC)
Palais des Congrès et des Expositions Nice Acropolis
Nice, France, December 11-13, 2019

Transfer Planning for Temporal Logic Tasks

Xusheng Luo and Michael M. Zavlanos

Abstract— **This paper proposes an optimal control synthesis algorithm for Linear Temporal Logic (LTL) tasks that exploits experience from solving similar LTL tasks before. The key idea is to appropriately decompose complex LTL tasks into simpler subtasks and define sets of skills, or plans, needed to solve these subtasks. These skills can be stored in a library of reusable skills and can be used to quickly synthesize plans for new tasks that have not been encountered before. Our proposed method is inspired by literature on multi-task learning and can be used to transfer experience between different LTL tasks. It amounts to a new paradigm in model-checking and optimal control synthesis methods that to this date do not use prior experience to solve planning problems. We present numerical experiments that show that our approach generally outperforms these methods in terms of time to generate feasible plans. We also show that our proposed algorithm is probabilistically complete and asymptotically optimal.**

I. Introduction

Control synthesis for mobile robots under complex tasks captured by Linear Temporal Logic (LTL) formulas has received considerable attention recently [1]–[8]. If optimality is not required, then model checking theory [9] can be used to find feasible paths that satisfy LTL-specified tasks [1]–[3]. Nevertheless, if optimality is of interest, then optimal control synthesis methods can be used to find the desired optimal plans. Such methods either draw from sampling-based motion planning algorithms in robotics [4]–[6] and can be probabilistically complete and asymptotically optimal, or formulate the LTL planning problem as an optimization problem [7], [8] that can be solved using available techniques. Common in the above works is that they solve every new planning problem from the beginning, without the use of any form of experience. However, most LTL tasks in robotics applications, even if they are defined by distinct LTL formulas, involve common basic operations such as simple reach-avoid subtasks. Therefore, it is meaningful to construct a library of reusable skills to speed up the search for feasible plans every time a new LTL task is encountered.

The key idea is to decompose a global LTL task into several subtasks, some of which may be used to later compose new tasks. The concept of reusing skills is more common in the machine learning literature. For example, transfer learning [10] employs prior knowledge to models which follow a different distribution from those in the training stage. Nevertheless, unlike transfer learning that is more focused on generalization, here the concept of reusing skills is more relevant to matching current tasks with skills in the library. Relevant is also work on hierarchical task planning [11] that adopts a divide and conquer strategy by recursively decomposing tasks into primitive action sequences. Hierarchical task planning (HTN) is a very general framework and it has been integrated with LTL in [12], where LTL is used to represent domain-specific control knowledge rather than specify required tasks, as in the problems considered here.

In this paper, we propose a transfer planning method that builds a library of skills that are as general as possible. Each skill can be viewed as a subtask. Inspired by the multi-task learning framework in [13], we design a path for each skill by constructing a set of subtrees in parallel using the optimal sampling-based method proposed in [6]. Then, every time a new LTL task is provided, we decompose it into several subtasks, and check whether these can be solved using skills stored in the library. We show our proposed algorithm is probabilistically complete and asymptotically optimal. Moreover, we compare our algorithm to the very fast biased sampling-based method in [14] and show that we outperform this method when sufficient number of skills can be reused to synthesize plans, especially for simpler tasks that require smaller numbers of skills to solve. Our approach amounts to a new paradigm in model-checking and optimal control synthesis methods that to this date do not use prior experience to solve planning problems.

II. Problem Formulation

Consider a robot operating in a workspace $\mathcal{W} \subset \mathbb{R}^d$, $d = 2, 3$. There are W disjoint regions $\{r_j\}_{j=1}^W$ of interest in $\mathcal{W}$. The workspace $\mathcal{W}$ can be represented by a weighted Transition System obtained through an abstraction process.

Definition 2.1 (TS): A *weighted Transition System* (TS), denoted by TS, is a tuple $\text{TS} = (\mathcal{X}, \mathbf{x}_0, \rightarrow_{\text{TS}}, d, \mathcal{AP}, L)$ where: (a) $\mathcal{X}$ is the set of discrete states; (b) $\mathbf{x}_0 \in \mathcal{X}$ is the initial position of the robot; (c) $\rightarrow_{TS} \subseteq \mathcal{X} \times \mathcal{X}$ is the transition relation; (d) $d : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}_+$ is a cost function that assigns a distance cost to each possible transition in TS; (e) $\mathcal{AP} = \bigcup_{j=1}^W \{\pi^{r_j}\}$ is the set of atomic propositions, where π^{r_j} is true if the robot is at region r_j and false otherwise; (f) $L : \mathcal{X} \rightarrow \mathcal{AP} \cup \{\pi^{\text{obs}}\} \cup \{\epsilon_\emptyset\}$ is an output function defined as $L(\mathbf{x}) = \pi^{r_j} | \pi^{\text{obs}} | \epsilon_\emptyset$, $\forall \mathbf{x} \in \mathcal{X}$, where π^{obs} represents obstacles, $\epsilon_\emptyset$ stands for empty output.

We define the *cost* of a finite path τ of form $\tau = \tau[0]\tau[1]\tau[2]\dots\tau[n]$ as $\hat{J}(\tau) = \sum_{k=0}^{n-1} d(\tau[k], \tau[k+1])$. Given an LTL task ϕ, we define the *language* $\texttt{Words}(\phi) = \{\sigma \in (2^{\mathcal{AP}})^\omega | \sigma \models \phi\}$, where $\models \subseteq (2^{\mathcal{AP}})^\omega \times \phi$ is the satisfaction relation, as the set of infinite words $\sigma \in (2^{\mathcal{AP}})^\omega$

Xusheng Luo and Michael M. Zavlanos are with the Department of Mechanical Engineering and Materials Science, Duke University, Durham, NC 27708, USA. {xusheng.luo, michael.zavlanos}@duke.edu. This work is supported in part by AFOSR under award #FA9550-19-1-0169 and by ONR under grant #N000141812374.

that satisfy ϕ. Any LTL formula ϕ can be translated into a Nondeterministic Büchi Automaton (NBA) as follows [9].

Definition 2.2 (NBA): A *Nondeterministic Büchi Automaton* (NBA) B over $2^{\mathcal{AP}}$ is defined as a tuple $B = \left(\mathcal{Q}_B, \mathcal{Q}_B^0, \Sigma, \rightarrow_B, \mathcal{Q}_B^F\right)$, where $\mathcal{Q}_B$ is the set of states, $\mathcal{Q}_B^0 \subseteq \mathcal{Q}_B$ is a set of initial states, $\Sigma = 2^{\mathcal{AP}}$ is an alphabet, $\rightarrow_B \subseteq \mathcal{Q}_B \times \Sigma \times \mathcal{Q}_B$ is the transition relation, and $\mathcal{Q}_B^F \subseteq \mathcal{Q}_B$ is a set of accepting/final states.

Given the TS and the NBA B, we define the *Product Büchi Automaton* (PBA) $P = \text{TS} \otimes B$, as follows [9]:

Definition 2.3 (PBA): Given the transition system $\text{TS} = (\mathcal{X}, \mathbf{x}_0, \longrightarrow_{\text{TS}}, d, \mathcal{AP}, L)$ and the NBA $B = (\mathcal{Q}_B, \mathcal{Q}_B^0, \Sigma, \rightarrow_B, \mathcal{Q}_B^F)$, we define the *Product Büchi Automaton* $P = \text{TS} \otimes B$ as a tuple $P = (\mathcal{Q}_P, \mathcal{Q}_P^0, \longrightarrow_P, \mathcal{Q}_P^F)$ where (a) $\mathcal{Q}_P = \mathcal{X} \times \mathcal{Q}_B$ is the set of states; (b) $\mathcal{Q}_P^0 = \mathbf{x}_0 \times \mathcal{Q}_B^0$ is a set of initial states; (c) $\longrightarrow_P \subseteq \mathcal{Q}_P \times 2^{\mathcal{AP}} \times \mathcal{Q}_P$ is the transition relation defined by the rule: $\frac{(\mathbf{x} \rightarrow_{\text{TS}} \mathbf{x}') \wedge (q_B \xrightarrow{L(\mathbf{x})} q_B')}{q_P = (\mathbf{x}, q_B) \longrightarrow_P q_P' = (\mathbf{x}', q_B')}$. Transition from state q_P to q_P' is denoted by $(q_P, q_P') \in \longrightarrow_P$; (d) $\mathcal{Q}_P^F = \mathcal{X} \times \mathcal{Q}_B^F$ is a set of accepting/final states.

We say that an infinite path τ of the form $\tau = \tau[0]\tau[1]\ldots$ satisfies ϕ if and only if $\texttt{trace}(\tau) \in \texttt{Words}(\phi)$, where $\texttt{trace}(\tau) = L(\tau[0])L(\tau[1])\ldots$. Specifically, a path satisfying ϕ can be written in the finite prefix-suffix structure, i.e., $\tau = \tau^{\text{pre}}[\tau^{\text{suf}}]^\omega = \Pi|_{\text{TS}} p^{\text{pre}}[\Pi|_{\text{TS}} p^{\text{suf}}]^\omega$, where the prefix part τ^{pre} is executed only once followed by the indefinite execution of the suffix part τ^{suf} and $\Pi|_{\text{TS}}$ stands for the projection of product space onto the state-space $\mathcal{X}$. The path p^{pre} starts from an initial state $q_P^0 \in \mathcal{Q}_P^0$ and ends at a final state $q_P^F \in \mathcal{Q}_P^F$. The path p^{suf} is a cycle around the above final state. Then, the goal is to minimize the following objective

$$J(\tau) = \beta \hat{J}(\tau^{\text{pre}}) + (1-\beta)\hat{J}(\tau^{\text{suf}}), \tag{1}$$

where $\hat{J}(\tau^{\text{pre}})$ and $\hat{J}(\tau^{\text{suf}})$ stand for the cost of the prefix and suffix part, respectively and $\beta \in [0,1]$ is a user-specified parameter. In this paper, we address the following problem.

Problem 1: Given a global LTL specification ϕ and a transition system TS, find a discrete plan τ that satisfies ϕ and minimizes the cost function (1).

III. Creation of the Library of Skills

To solve Problem 1 we propose a new planning method that exploits experience from solving similar planning problems before. In this section, we first propose an algorithm to decompose an LTL task into simpler subtasks that are executed sequentially, therefore, giving rise to a graph G_T of subtasks. Then, for each subtask, we find a plan by growing multiple subtrees in parallel using the method in [6]. These plans constitute the skills stored in the library $\mathfrak{L}$.

A. Construction of the graph of subtasks

Given a formula ϕ and the NBA B, consider the transitions $q_B^i(\sigma_i) \xrightarrow{\pi^{r_i}}_B q_B^j(\sigma_j) \xrightarrow{\pi^{r_j}}_B q_B^k(\sigma_k)$, where σ represents the label of the self-loop around the Büchi state. Suppose that currently the Büchi state is q_B^i and the robot is at location r_i so π^{r_i} is true, enabling the transition to q_B^j. To enable the next transition, the robot should visit r_j and the path should satisfy the label σ_j. We denote this subtask by the tuple $((\pi^{r_i}, q_B^i), (\pi^{r_j}, q_B^j))$ and is formally defined as follows.

Algorithm 1: $\texttt{GraphofSubtasks}(B, \text{TS})$

$\mathcal{S}_f = \emptyset, \mathcal{S}_e = \emptyset$;
$\mathcal{S}_f \leftarrow ((\mathbf{x}_0, q_B^0), \epsilon)$, $\mathcal{R}_B^T(((\mathbf{x}_0, q_B^0), \epsilon)) = q_B^0$;
while $\mathcal{S}_f \neq \emptyset$ **do**
  $q_T^{\text{curr}} = \texttt{GetNode}(\mathcal{S}_f)$, $q_B^{\text{target}} = \mathcal{R}_B^T(q_T^{\text{curr}})$;
  for $q_B^{succ} \in \mathcal{R}_B(q_B^{target}) \setminus \{q_B^{target}\}$ **do**
    $\sigma_e = \texttt{GetLabel}(q_B^{\text{target}}, q_B^{\text{succ}})$;
    $\mathcal{D} = \texttt{GetDestination}(\sigma_e)$;
    for $\mathbf{x} \in \mathcal{D}$ **do**
      $q_P = (\mathbf{x}, q_B^{\text{target}})$;
      if $\sim \texttt{Match}(\mathcal{E}_T, \Pi|_P q_T^{curr}, q_P)$ **then**
        $\mathcal{E}_T \leftarrow (\Pi|_P q_T^{\text{curr}}, q_P)$;
      $\mathcal{R}_B^T((q_P, \sigma_e)) = q_B^{\text{succ}}$;
      if $(q_P, \sigma_e) \notin (\mathcal{S}_f \cup \mathcal{S}_e)$ **then**
        $\mathcal{S}_f \leftarrow (q_P, \sigma_e)$, $\mathcal{V}_T \leftarrow q_P$;
  $\mathcal{S}_e \leftarrow q_T^{\text{curr}}$;

Definition 3.1 (Subtask): A subtask $((\mathbf{x}_s, q_B^s), (\mathbf{x}_e, q_B^e))$ is a tuple where $(\mathbf{x}_s, \mathbf{x}_e) \in \rightarrow_{TS}$, $(q_B^s, L(\mathbf{x}_s), q_B^e) \in \rightarrow_B$ and there exists another Büchi state q_B such that $(q_B^e, L(\mathbf{x}_e), q_B) \in \rightarrow_B$. We call $q_P^s = (\mathbf{x}_s, q_B^s)$ the starting endpoint and $q_P^e = (\mathbf{x}_e, q_B^e)$ the terminal endpoint.

A subtask defines a set of paths that connect $\mathbf{x}_s$ and $\mathbf{x}_e$ while satisfying the label of the self-loop around q_B^e. Next, we build a graph of subtasks $G_T = (\mathcal{V}_T, \mathcal{E}_T)$, where $\mathcal{V}_T = \{\mathbf{x}_0, q_B^0\} \cup \{(\mathbf{x}, q_B) \,|\, L(\mathbf{x}) \in \mathcal{AP}, q_B \in \mathcal{Q}_B\}$ denotes the node set and $\mathcal{E}_T = \{((\mathbf{x}_s, q_B^s), (\mathbf{x}_e, q_B^e)) | \mathbf{x}_s \rightarrow_{\text{TS}} \mathbf{x}_e, q_B^s \xrightarrow{L(\mathbf{x}_s)}_P q_B^e\}$ denotes the edge set. The graph of subtasks can be viewed as an abstraction of $\mathcal{Q}_P$ and it captures high-level task dynamics. We construct the graph of subtasks G_T using Alg. 1, which we describe next.

Alg. 1 takes as input the NBA B and the transition system TS, conducts a search over B and returns the graph G_T. First, B is pruned by removing infeasible labels that require the robot to be at two disjoint regions simultaneously; and details about pruning can be found in [14]. Alg. 1 selects one state from the set $\mathcal{S}_f$, then expands it and stores its direct successors in the set $\mathcal{S}_f$ for further expansion.[1] After expansion, this state is saved in the expanded set $\mathcal{S}_e$. Each state in these two sets is in the form of $((\mathbf{x}, q_B), \sigma)$, where $(\mathbf{x}, q_B)$ will be a node in G_T and σ is an label. Note that for a state $(\mathbf{x}, q_B)$, q_B can nondeterministically transition to one of its multiple direct successors, since $L(\mathbf{x})$ satisfies possibly more than one labels. But the direct successor of q_B that corresponds to the associated label σ is unique. We define a lookup table $\mathcal{R}_B^T$ that outputs the direct successor of q_B when a node $(\mathbf{x}, q_B)$ and the label σ are provided.

Alg. 1 is initialized by creating an artificial transition $(q_B^0, q_B^0) \rightarrow_B$, where $\leftarrow$ means adding the right-hand side element to the left-hand side set. The construction of G_T

[1] A direct successor (precessor) means that two nodes are connected by an edge, while successors (precessors) mean a path exists between two nodes.

proceeds by selecting a state q_T^{curr} from $\mathcal{S}_f$ per iteration and denoting its direct successor as $q_B^{\text{target}} = \mathcal{R}_B^T(q_T^{\text{curr}})$ [Alg. 1, line 4]. Next, let $\mathcal{R}_B(q_B^{\text{target}})$ denote the set of direct successors of q_B^{target} in B, and we generate a set of subtasks for every state in $\mathcal{R}_B(q_P^{\text{target}})$, if such states exist, excluding q_B^{target} itself [Alg. 1, line 5-14]. We ignore subtasks with identical endpoints since they can be trivially fulfilled. For $q_B^{\text{succ}} \in \mathcal{R}_B(q_B^{\text{target}})$, we first obtain the label σ_e such that $q_B^{\text{target}} \xrightarrow{\sigma_e}_B q_B^{\text{succ}}$ using the function `GetLabel`. Assuming it is in disjunctive normal form, i.e., $\sigma_e = \vee_{\forall r_i}(\wedge_{\forall r_j \neq r_i} \neg \pi^{r_j} \wedge \pi^{r_i})$, then, a set $\mathcal{D}$ of destinations, one destination per conjunction, that satisfies σ_e is obtained by calling the function `GetDestination`. Then, for every $\mathbf{x} \in \mathcal{D}$, we check whether a new subtask (that has not yet been seen) can be created [Alg. 1, line 8-14]. Specifically, denoting by $\Pi|_P$ the projection onto the product state space, we define by $\Pi|_P q_T^{\text{curr}}$ the starting endpoint and let $q_P = (\mathbf{x}, q_B^{\text{target}})$ be a corresponding terminal endpoint. In [Alg. 1, line 10-11], we check whether an equivalent subtask has been created before, and we discuss how to decide the equivalence between two subtasks later. If no equivalent subtask exists, an edge corresponding to this new subtask is added to the graph G_T and the lookup table $\mathcal{R}_B^T$ is updated [Alg. 1, line 12]. If (q_P, σ_e) does not belong to either $\mathcal{S}_f$ or $\mathcal{S}_e$, we add it to $\mathcal{S}_f$ and add q_P to the node set $\mathcal{V}_T$ [Alg. 1, line 13-14]. Finally, when all candidate direct successors have been considered, q_T^{curr} is added to the expanded set $\mathcal{S}_e$.

In what follows, we discuss how to check if two subtasks are equivalent. Assuming that the path connecting any two locations is reversible, i.e., the path from A to B is the path from B to A, we define equivalent subtasks as follows.

Definition 3.2 (Equivalent subtasks): Two subtasks, denoted by (q_P^s, q_P^e) and $(\tilde{q}_P^s, \tilde{q}_P^e)$, are equivalent if

(i) They have the same pair of initial and terminal positions, i.e., $(\tilde{\mathbf{x}}_s = \mathbf{x}_s \wedge \tilde{\mathbf{x}}_e = \mathbf{x}_e) \vee (\tilde{\mathbf{x}}_s = \mathbf{x}_e \wedge \tilde{\mathbf{x}}_e = \mathbf{x}_s)$;

(ii) The labels around the second Büchi states are the same, i.e., $\texttt{GetLabel}(q_B^e, q_B^e) = \texttt{GetLabel}(\tilde{q}_B^e, \tilde{q}_B^e)$.

We refer to the first conjunction in (i) as "forward match" and the second as "backward match".

B. Construction of reusable skills

After obtaining a pool of subtasks, we design paths for each subtask to include in a lookup library $\mathfrak{L}$. To this end, we incrementally build multiple subtrees in parallel with every subtree being constructed using the sampling-based method in [6]. Specifically, we first divide all subtasks into different groups based on starting endpoints. Every group collects subtasks with the same starting endpoint and is associated with a subtree. The root of this subtree corresponds to the starting endpoint of that group. Every subtree is responsible for finding one path for every subtask in its group. These subtrees are grown independently. First, a state is sampled from the $\mathcal{Q}_P$. If this state can be reached from any one of subtrees, then we check whether any of the roots of the other subtrees can be reached via this state. Since these subtrees are grown in parallel, feasible plans can be found much faster; see also Alg. 2. Alg. 2 initializes a set of subtrees

Algorithm 2: `BuildLibrary`$(G_T, \mathcal{S}_r)$

```
Initialize a set {T_k = (V_k, E_k)} of subtrees with each
    root q_P^{r_k} ∈ S_r;
while Σ_{i=1}^{|{T_k}|} |V_i| < n_max do
    x_new = Sample(T_{Uniform(1,|{T_k}|)});
    for i = 1 : 1 : |{T_k}| do
        for q_B ∈ Q_B do
            q_P^new = (x_new, q_B);
            if q_P^new ∉ V_i then
                Extend(T_i, q_P^new, R_P^→(q_P^new) \ R_T(q_P^{r_i})) ;
            else
                Rewire(T_i, q_P^new, R_P^←(q_P^new));
                ConnectRoots(T_i, R_T(q_P^{r_i}), q_P^new);
Extract paths for subtasks and build the library 𝔏 ;
```

$\{\mathcal{T}_k = (\mathcal{V}_k, \mathcal{E}_k)\}$ with roots $\mathcal{S}_r = \{q_P^s \,|\, (q_P^s, q_P^e) \in \mathcal{E}_T\}$ [Alg. 2, line 1]. Let $q_P^{r_k}$ denote the root of k-th subtree. Next, the parallel construction grows each subtree until the total number of nodes exceeds a user-defined number $n_{\max}$.

1) Sampling the position component $\mathbf{x}_{new}$: At the beginning of each iteration, the function `Sample` samples a new location $\mathbf{x}_{\text{new}}$ [Alg. 2, line 3] by randomly sampling a subtree, then randomly sampling a node q_P in that subtree and, finally, sampling a location $\mathbf{x}_{\text{new}}$ that can be reached from $\Pi|_{\text{TS}} q_P$, i.e, $\Pi|_{\text{TS}} q_P \rightarrow_{\text{TS}} \mathbf{x}_{\text{new}}$. $\mathbf{x}_{\text{new}}$ is paired with every $q_B \in \mathcal{Q}_B$ to create a new product state $q_P^{\text{new}} = (\mathbf{x}_{\text{new}}, q_B)$. Then we check whether it can be reached by any of the subtrees using functions `Extend` and `Rewire`.

2) Extending towards q_P^{new} in $\mathcal{T}_i$: If $q_P^{\text{new}} \notin \mathcal{V}_i$, the `Extend` step tries to add q_P^{new} to tree $\mathcal{T}_i$. Let $\mathcal{R}_P^{\rightarrow}(q_P^{\text{new}})$ collect nodes in $\mathcal{T}_i$ that can reach q_P^{new} in one hop and $\mathcal{R}_T(q_P^{r_i})$ denote the direct successors of $q_P^{r_i}$ in $\mathcal{V}_T$. Similar to [6], the function `Extend` checks all nodes in $\mathcal{R}_P^{\rightarrow}(q_P^{\text{new}})$ to find the shortest path leading to q_P^{new}, with the exception that nodes of $\mathcal{R}_T(q_P^{r_i})$ are excluded from $\mathcal{R}_P^{\rightarrow}(q_P^{\text{new}})$ [Alg. 2, line 7-8]. Since we check all subtrees, if there exists one root $q_P^{r_j} \in \mathcal{R}_T(q_P^{r_i})$ that is already in the subtree $\mathcal{T}_i$ and it can reach q_P^{new}, q_P^{new} will be added to the subtree with root $q_P^{r_j}$. Thus, there is no need to connect q_P^{new} to $q_P^{r_j}$ in the current subtree $\mathcal{T}_i$. In this case, $q_P^{r_j}$ does not have outgoing edges.

3) Rewiring through q_P^{new} in $\mathcal{T}_i$: If $q_P^{\text{new}} \in \mathcal{V}_i$, the `Rewiring` step checks whether q_P^{new} can improve the cost of other nodes. Let $\mathcal{R}_P^{\leftarrow}(q_P^{\text{new}})$ be the set that collects all nodes in $\mathcal{T}_i$ that q_P^{new} can reach in one hop. The function `Rewire` is identical to that in [6] which checks all nodes in $\mathcal{R}_P^{\leftarrow}(q_P^{\text{new}})$ in case there exists a shorter path leading to them that passes through q_P^{new}. If any node q_P is successfully rewired, the cost of all its successors will be updated.

4) Extending towards other successive roots from $\mathcal{T}_i$: This steps checks whether q_P^{new} builds a connection between the current root $q_P^{r_i}$ (the starting endpoint of one subtask) and its direct successor $q_P^{\text{end}} \in \mathcal{R}_T(q_P^{r_i})$ (the terminal endpoint); see Alg. 3. The algorithm extends $\mathcal{T}_i$ towards it or rewire through q_P^{new}. If q_P^{end} is not in the current tree and q_P^{new} can reach q_P^{end}, then q_P^{end} is added to $\mathcal{T}_i$ [Alg. 3, line 3-5], where

Algorithm 3: $\texttt{ConnectRoots}(\mathcal{T}_i, \mathcal{R}_T(q_P^{r_i}), q_P^{\text{new}})$

for $q_P^{end} \in \mathcal{R}_T(q_P^{r_i})$ **do**
  $c = \texttt{cost}(\mathcal{T}_i, q_P^{\text{new}}) + d(\Pi|_{\text{TS}} q_P^{\text{new}}, \Pi|_{\text{TS}} q_P^{\text{end}})$;
  if $q_P^{end} \notin \mathcal{V}_i \wedge (q_P^{new}, q_P^{end}) \to_P$ **then**
    $\mathcal{V}_i \leftarrow q_P^{\text{end}}$, $\mathcal{E}_i \leftarrow (q_P^{\text{new}}, q_P^{\text{end}})$;
    $\texttt{cost}(\mathcal{T}_i, q_P^{\text{end}}) = c$, $\mathcal{S}_i \leftarrow q_P^{\text{end}}$;
  else if $q_P^{end} \in \mathcal{V}_i \wedge \texttt{cost}(\mathcal{T}_i, q_P^{end}) > c \wedge (q_P^{new}, q_P^{end}) \to_P$ **then**
    $\mathcal{E}_i = \mathcal{E}_i \setminus \{(\texttt{Parent}(q_P^{\text{end}}), q_P^{\text{end}})\}$;
    $\mathcal{E}_i \leftarrow (q_P^{\text{new}}, q_P^{\text{end}})$, $\texttt{cost}(\mathcal{T}_i, q_P^{\text{end}}) = c$;

$\texttt{cost}(\mathcal{T}_i, q_P^{\text{end}})$ represents the distance from the root $q_P^{r_i}$ to q_P^{end}. Note that q_P^{end} might be the root of another subtree, but any operation performed in $\mathcal{T}_i$ does not affect that subtree. We add q_P^{end} to a set $\mathcal{S}_i$ which keeps track of subtasks for which a feasible solution of the subtask $(q_P^{r_i}, q_P^{\text{end}})$ is found in $\mathcal{T}_i$. If it is the case that q_P^{end} belongs to $\mathcal{V}_i$, q_P^{new} can transition to q_P^{end} and incurs a shorter path, the predecessor of q_P^{end} is changed to q_P^{new} and the cost is updated [Alg. 3, line 6-8].

5) Extracting paths for subtasks: After the iteration terminates, Alg. 2 extracts one path of product states for each subtask in $\mathcal{S}_i$ (which appears in Alg. 3, line 5) and saves them in the library $\mathfrak{L}$ [Alg. 2, line 12]. For any $q_P^{\text{end}} \in \mathcal{S}_i$, we obtain the path by tracing back from q_P^{end} to the root $q_P^{r_i}$.

IV. Control synthesis for new formulas

In this section, we generate a path for a new formula by reusing skills from the library $\mathfrak{L}$. First, a graph of subtasks for the new formula is constructed following almost the same algorithm as Alg. 1. Then, the graph of subtasks is matched with the library, creating a lookup table of skills $\mathfrak{L}_{\text{reuse}} \subseteq \mathfrak{L}$ that can be reused for the new formula. Finally, a tree is constructed to find the feasible paths for the new formula.

A. Construction of new subtasks and matching with skills

Since we need to find a feasible path for the new formula ϕ, we can't employ Alg. 1 directly, but the modifications are minor. We first build a new graph of subtasks $\tilde{G}_T = (\tilde{\mathcal{V}}_T, \tilde{\mathcal{E}}_T)$ for ϕ without checking whether any two subtasks in $\tilde{G}_T$ are equivalent, i.e., adding the edge directly to $\tilde{G}_T$ without calling the `Match` function; see Alg. 1, line 10-11. After $\tilde{G}_T$ is constructed, we check whether any of subtasks can be matched with any skill in the library. To make the most use of these skills, we relax the second requirement in the definition of equivalent tasks. Instead, for $(\tilde{q}_P^s, \tilde{q}_P^e) \in \tilde{\mathcal{E}}_G$ and $(q_P^s, q_P^e) \in \mathfrak{L}$, if the path of (q_P^s, q_P^e) satisfies the label around $\tilde{q}_B^e$, then we claim a match. We can't use the skills directly, since $(\tilde{q}_P^s, \tilde{q}_P^e)$ and (q_P^s, q_P^e) have different Büchi states due to different LTL formulas. The next step is to replace the Büchi components of the matched path with the new Büchi components. Suppose the path corresponding to (q_P^s, q_P^e) is $(\mathbf{x}_s, q_B^s), (\mathbf{x}_2, q_B^e), \ldots, (\mathbf{x}_e, q_B^e)$,[2]

[2]Since $\mathbf{x}_s$ makes the transition from q_B^s to q_B^e occur, the second state in any path that satisfies the subtask (q_P^s, q_P^e) is in the form of $(\cdot, q_B^e)$, and the rest states of this path have the same Büchi component q_B^e.

Algorithm 4: $\texttt{TransferPlanning}(B, \mathcal{S}_r, \mathfrak{L}_{\text{reuse}}, n_{\max})$

Initialize tree $\mathcal{T}$ with $\texttt{cost}(q_P^0) = 0$ and `acc` ;
$\texttt{ConnectReusedPath}(q_P^0, \texttt{Reuse}(\mathcal{T}, q_P^0, \mathfrak{L}_{\text{reuse}}))$;
while $|\mathcal{V}| < n_{\max}$ **do**
  $\mathbf{x}_{\text{new}} = \texttt{Sample}(\mathcal{T})$;
  for $q_B \in \mathcal{Q}_B$ **do**
    $q_P^{\text{new}} = (\mathbf{x}_{\text{new}}, q_B)$;
    if $q_P^{new} \notin \mathcal{V}$ **then**
      $\texttt{Extend}(\mathcal{T}, q_P^{\text{new}}, \mathcal{R}_P^{\rightarrow}(q_P^{\text{new}}))$;
    if $q_P^{new} \in \mathcal{V}$ **then**
      $\texttt{Rewire}(\mathcal{T}, q_P^{\text{new}}, \mathcal{R}_P^{\leftarrow}(q_P^{\text{new}}))$;
      $\texttt{FindGoal}(\mathcal{T}, q_P^{\text{new}})$;
      $\mathfrak{L}'_{\text{reuse}} \leftarrow \texttt{Reuse}(\mathcal{T}, q_P^{\text{new}}, \mathfrak{L}_{\text{reuse}})$;
      $\texttt{ConnectStates}(\mathcal{T}, \mathcal{S}_r, q_P^{\text{new}})$;
  $\texttt{ConnectReusedPath}(\mathcal{T}, q_P^{\text{new}}, \mathfrak{L}'_{\text{reuse}})$;
$\texttt{FeasiblePath}(\mathcal{T}, \mathcal{Q}_{\mathcal{T}}^{\text{goal}})$;

(i) if a forward match takes place, the path for the current subtask is replaced by $(\mathbf{x}_s, \tilde{q}_B^s), (\mathbf{x}_2, \tilde{q}_B^e), \ldots, (\mathbf{x}_e, \tilde{q}_B^e)$;
(ii) if a backward match takes place, the path for the current subtask is replaced by $(\mathbf{x}_e, \tilde{q}_B^s), \ldots, (\mathbf{x}_2, \tilde{q}_B^e), (\mathbf{x}_s, \tilde{q}_B^e)$, i.e., the path is reversed first and then replaced.

For every subtask that a skill is matched, both the subtask and the path are saved in library $\mathfrak{L}_{\text{reuse}}$, which is a subset of $\mathfrak{L}$ consisting of skills that can be reused in the current task. If none is matched, we add $\tilde{q}_P^s$ to a set $\mathcal{S}_r$ for future use.

B. Finding a new feasible path

In this part, we build a tree incrementally to find feasible paths for the new formula with the root being the initial state q_P^0; see Alg. 4. The root is initialized with $\texttt{cost} = 0$.

Different from [6], we find the prefix and suffix parts simultaneously by associating each node with a second property `acc`, which stores the accepting states in $\mathcal{Q}_P^F$ that are in the path from q_P^0 to this node, i.e., $\texttt{acc}(q_P) = \{q'_P \,|\, q'_P \in \mathcal{Q}_P^F, \forall q'_P \in \texttt{Path}(q_P)\}$. After initialization, Alg. 4 first connects reusable paths in $\mathfrak{L}_{\text{reuse}}$ that can be connected to the root. This part is discussed in Section IV-B.4. Then, Alg. 4 alternates among connecting to the sampled state and reusable skills, until the terminal criterion is reached.

1) Extending towards q_P^{new} and rewiring through q_P^{new}: If $q_P^{\text{new}} \notin \mathcal{V}$, the `Extend` step here is exactly the same as that in Section III-B.2, with the additional operation of updating the property `acc` [Alg. 4, line 8]. Assuming that q_P^{new} is an accepting state and that its parent is q_P^{parent}, then $\texttt{acc}(q_P^{\text{new}}) = \texttt{acc}(q_P^{\text{parent}}) \cup \{q_P^{\text{new}}\}$, else $\texttt{acc}(q_P^{\text{new}}) = \texttt{acc}(q_P^{\text{parent}})$.[3] If $q_P^{\text{new}} \in \mathcal{V}$, the `Rewire` step follows the same procedure as that in Section III-B.3, meanwhile the property `acc` of the successors of these nodes that their parents are changed to q_P^{new} is updated [Alg. 4, line 10].

2) Determining the goal state: We define the goal state as a triplet $(q_P, q_P^{\text{mid}}, q_P^F)$ if q_P^{mid} and q_P^F are in the path from q_P^0 to q_P, and q_P can reach q_P^{mid} in one hop then

[3]The right way to update `acc` is to monitor the accepting states along the path to q_P. Here, we only consider `acc` of its parent. The simplication does not affect the property, since `acc` is updated very often.

Algorithm 5: $\texttt{Reuse}(\mathcal{T}, q_P^{\text{new}}, \mathfrak{L}_{\text{reuse}})$

$\mathfrak{L}'_{\text{reuse}} = \emptyset$;
for $q_P^s \in \mathcal{K}_1(\mathfrak{L}_{reuse})$ **do**
  if $(q_P^{new}, q_P^s) \rightarrow_P \vee\, q_P^{new} = q_P^s$ **then**
    if $\texttt{Used}(q_P^s) =$*True* **then**
      $\mathfrak{L}'_{\text{reuse}} = [\mathfrak{L}'_{\text{reuse}}, (q_P^{\text{new}}, q_P^s)]$;
    else
      for $(q_P^s, q_P^e) \in \mathcal{K}_2(q_P^s)$ **do**
        $\mathfrak{L}'_{\text{reuse}} = [\mathfrak{L}'_{\text{reuse}}, (q_P^{\text{new}}, \mathfrak{L}_{\text{reuse}}(q_P^s, q_P^e))]$;
        Recursively add activated reusable paths to $\mathfrak{L}'_{\text{reuse}}$ and mark respective starting endpoints as "used";
      $\texttt{Used}(q_P^s) =$ True;

q_P^{mid} can reach q_P^F in multiple hops. The path from q_P^0 to q_P^F of the form $q_P^0, \ldots, q_P^{\text{mid}}, \ldots, q_P^F$ is the prefix part, and the path of the form $q_P^F, \ldots, q_P, q_P^{\text{mid}}, \ldots, q_P^F$ is the suffix part. We denote the set of goal states by $\mathcal{Q}_P^{\text{goal}} = \{(q_P, q_P^{\text{mid}}, q_P^F) \,|\, (q_P, q_P^{\text{mid}}) \rightarrow_P, q_P^{\text{mid}} \in \texttt{Path}(q_P), q_P^F \in \mathcal{Q}_P^F, q_P^F \in \texttt{Path}[q_P^{\text{mid}}, q_P]\}$, where $\texttt{Path}[q_P^{\text{mid}}, q_P]$ denotes the path between q_P^{mid} and q_P. If q_P^{new} is in the tree, then the function `FindGoal` updates goal set $\mathcal{Q}_{\mathcal{T}}^{\text{goal}}$ by determining whether q_P^{new} generates a goal state [Alg. 4, line 11].

3) Extending towards states in $\mathcal{S}_r$ [Alg. 4, line 13]: When q_P^{new} is added to $\mathcal{T}$, we check whether $\mathcal{T}$ can reach states in $\mathcal{S}_r$ via q_P^{new}. If $q_P^s \in \mathcal{S}_r$ is not in $\mathcal{V}$ and q_P^{new} can extend towards q_P^s, we add q_P^s to $\mathcal{T}$ through q_P^{new}. Otherwise if $q_P^s \in \mathcal{S}_r$ is in $\mathcal{V}$ and q_P^{new} can extend towards q_P^s, we check whether q_P^{new} can incur a smaller cost. If yes, we change the parent of q_P^s to q_P^{new} and update `cost` and `acc` of it and its successors.

4) Extending towards reusable skills [Alg. 4, line 14]: We extend q_P^{new} to possible reusable skills ; see Alg. 6. The first step is to get a library $\mathfrak{L}'_{\text{reuse}}$ consisting of all reusable skills in $\mathfrak{L}_{\text{reuse}}$ that q_P^{new} can connect to directly or indirectly [Alg. 4, line 12]. The details are in Alg. 5. Let $\mathcal{K}_1(\mathfrak{L}_{\text{reuse}})$ denote the set of starting endpoints of all subtasks, i.e., $\mathcal{K}_1(\mathfrak{L}_{\text{reuse}}) = \{q_P^s \,|\, (q_P^s, q_P^e) \in \mathfrak{L}_{\text{reuse}}\}$. Similarly, $\mathcal{K}_2(q_P^s)$ collects all subtasks in $\mathfrak{L}_{\text{reuse}}$ with starting endpoint q_P^s, i.e., $\mathcal{K}_2(q_P^s) = \{(q_P^s, q_P^e) \in \mathfrak{L}_{\text{reuse}}\}$. Alg. 5 checks every starting endpoint $q_P^s \in \mathcal{K}_1(\mathfrak{L}_{\text{reuse}})$ to find the largest set of reusable skills that can be connected to q_P^{new} in a direct or indirect way [Alg. 5, line 3-9]. If q_P^{new} can reach q_P^s in one hop or q_P^{new} is equivalent to q_P^s, then the path starting with q_P^s can be used here. $\texttt{Used}(q_P^s)$ indicates whether the reusable paths starting with q_P^s have been used. If true, we just add the pair to $\mathfrak{L}'_{\text{reuse}}$ [Alg. 5, line 5]. Else, for $(q_P^s, q_P^e) \in \mathcal{K}_2(q_P^s)$, we add the corresponding path to $\mathfrak{L}'_{\text{reuse}}$ with q_P^{new} inserted in the first place, which marks the state that q_P^s is connected to [Alg. 5, line 8]. Then, all indirectly reusable paths that can be connected to q_P^{new} due to direct or other indirect paths are added recursively, and respective starting endpoints are marked as "used". We refer to this use as "greedy use".

Note that each element in $\mathfrak{L}'_{\text{reuse}}$ is a path. In Alg. 6, we check every path $p \in \mathfrak{L}'_{\text{reuse}}$ returned by Alg. 5 and add it to the tree $\mathcal{T}$. For each state $q_P^{\text{reuse}} \in p$ except the first one, if q_P^{reuse} doesn't belong to the tree $\mathcal{T}$, q_P^{reuse} is added

Algorithm 6: $\texttt{ConnectReusedpath}(\mathcal{T}, \mathfrak{L}'_{\text{reuse}})$

for $p \in \mathfrak{L}'_{reuse}$ **do**
  for $q_P^{reuse} \in p[1 : end]$ **do**
    if $q_P^{reuse} \notin \mathcal{V}$ **then**
      $\texttt{Extend}(\mathcal{T}, q_P^{\text{reuse}}, \{\texttt{Parent}(p, q_P^{\text{reuse}})\})$;
    else
      $\texttt{Extend}(\mathcal{T}, q_P^{\text{reuse}}, \{\texttt{Parent}(\mathcal{T}, q_P^{\text{reuse}}), \texttt{Parent}(p, q_P^{\text{reuse}})\})$;
    $\texttt{FindGoal}(\mathcal{T}, q_P^{\text{reuse}})$;

to $\mathcal{T}$ through its parent in p [Alg. 6, line 4]. If q_P^{reuse} is in $\mathcal{T}$, we compare its parent in $\mathcal{T}$ with its parent in p to see which incurs a smaller cost [Alg. 6, line 6] Here, the function `Extend` is the same as that in Alg. 4, line 8. Next, we check whether q_P^{reuse} is a goal state [Alg. 6, line 7].

5) Finding the path leading to $q_P \in \mathcal{V}$: After Alg. 4 terminates, we extract a feasible path for each goal state in $\mathcal{Q}_{\mathcal{T}}^{\text{goal}}$ (The path may not exist due to the simplified update of `acc`). Given a path p in the product space, we can obtain a path τ in the workspace by projection. Then we can compute the cost and obtain the best feasible path τ.

V. Correctness and Optimality

Theorem 5.1: Assume that there exists a feasible plan τ satisfying the desired formula ϕ. Then, Alg. 4 is probabilistically complete and asymptotically optimal.

Proof: Observe that without the reuse of skills, Alg. 4 builds a tree which grows according to the same sampling-based method in [6]. The sampled states will eventually cover the whole finite product space. Reusing skills serves in accelerating the detection of a solution. The way to find the prefix and suffix parts at the same time does not affect the operations `Extend` and `Rewire`. Thus, following the proof of Thm. 5.9 and 5.11 in [6], we can maintain the property of probabilistic completeness and asymptotic optimality. ■

VI. Numerical Experiments

In this section, we present two case studies, implemented using Python 3.6.3 on a computer with 2.3 GHz Intel Core i5 and 8G RAM, that illustrate the efficiency of the proposed algorithm. The 20×20 grid workspace has 6 regions and 13 obstacles. The initial position of the robot is $(0.025, 0.025)$.

A. Transfer planning for different LTL tasks

We first use the formula $\Diamond\pi^{r_1} \wedge \Diamond\pi^{r_2} \wedge \Diamond\pi^{r_3} \wedge \Diamond\pi^{r_4} \wedge \Diamond\pi^{r_5} \wedge \Diamond\pi^{r_6}$ to create a library of 21 skills which includes paths between any two regions and paths from the initial location to regions. We test the following 6 formulas

$\phi_1 = \Box(\Diamond\pi^{r_1} \wedge \Diamond\pi^{r_2})$
$\phi_2 = \Box(\Diamond\pi^{r_1} \wedge \Diamond\pi^{r_4}) \wedge \Box(\pi^{r_1} \rightarrow \bigcirc(!\pi^{r_1} U \pi^{r_4}))$
$\phi_3 = \Box(\Diamond\pi^{r_1} \wedge \Diamond\pi^{r_2} \wedge \Diamond\pi^{r_3} \wedge \Diamond\pi^{r_4} \wedge \Diamond\pi^{r_5} \wedge \Diamond\pi^{r_6})$
$\phi_4 = \Box\Diamond(\pi^{r_1} \wedge \Diamond\pi^{r_2}) \wedge \Box\Diamond(\pi^{r_3} \wedge \Diamond\pi^{r_4}) \wedge !\pi^{r_1} U \pi^{r_3}$
$\phi_5 = \Box(\Diamond\pi^{r_1} \wedge \Diamond\pi^{r_2}) \wedge \Box(\Diamond\pi^{r_3} \wedge \Diamond\pi^{r_4}) \wedge \Diamond\pi^{r_5} \wedge \Diamond\pi^{r_6}$
$\quad \wedge \Box(\pi^{r_1} \rightarrow \bigcirc(!\pi^{r_1} U \pi^{r_2})) \wedge \Box(\pi^{r_3} \rightarrow \bigcirc(!\pi^{r_3} U \pi^{r_4}))$
$\phi_6 = \Box(\Diamond\pi^{r_1} \wedge \Diamond\pi^{r_2}) \wedge \Box(\Diamond\pi^{r_3} \wedge \Diamond\pi^{r_4}) \wedge \Box(\Diamond\pi^{r_5} \wedge \Diamond\pi^{r_6})$
$\quad \wedge \Box(\pi^{r_1} \rightarrow \bigcirc(!\pi^{r_1} U \pi^{r_2})) \wedge \Box(\pi^{r_3} \rightarrow \bigcirc(!\pi^{r_3} U \pi^{r_4}))$

TABLE I: Runtimes and costs for different LTL tasks

tasks	$t_{\mathfrak{L}}$	$\#T$	$\%T$	t	t [14]	J	J [14]
ϕ_1	0.00	6	83.3	0.02	0.22	1.85	1.86
ϕ_2	0.03	17	94.1	0.01	0.20	1.60	1.47
ϕ_3	0.02	10	90.0	0.01	0.70	3.40	3.45
ϕ_4	0.06	57	91.2	0.07	1.16	3.25	3.15
ϕ_5	2.04	383	94.3	0.20	3.40	5.10	2.83
ϕ_6	9.10	1018	90.5	0.98	9.85	4.50	3.67

$t_{\mathfrak{L}}$ is the time to build the graph G_T of subtasks and identify matching skills, $\#T$ is the number of subtasks in G_T, $\%T$ is the percentage of subtaks that have reusable paths, t and t [14] are the runtimes in seconds to find the first feasible path using transfer planning and the biased sampling-based method in [14], J and J [14] are the lengths of the path found. $\beta = 0.5$ in (1).

$$\wedge \square(\pi^{r_5} \rightarrow \bigcirc(!\pi^{r_5} U \pi^{r_6}))$$

and compare with the biased sampling-based method in [14] in terms of runtime and path quality. We run 10 trials for each task and the averaged results are shown in Tab. I.

As Tab. I shows, the basic skills are representative, with use ratio more than 80%. Our algorithm can find the first feasible path in a shorter time. Due to the greedy use in Alg. 5, reducing cost is not a priority, even though the optimal path can be found after running enough iterations. The time to build the graph of subtasks for the new formula and identify matching subtasks from the library increases with the complexity of the task. For simple formulas our algorithm performs better than [14] but for complicated ones it performs worse. In the current version of our algorithm, computational complexity depends on the size of G_T.

B. Transfer planning for different environments

We assume that the environment changes slightly after a path p is found. Instead of planning from the beginning, we modify the original path by discovering subtasks that require replanning. Specifically, we assume that $p = (\mathbf{x}_0, q_B^0), (\mathbf{x}_1, q_B^1), \ldots, (\mathbf{x}_n, q_B^n)$ and that the change is captured by the presence of m new obstacles in place of some waypoints or locations between two waypoints in the path p. Obstacles do not replace regions. We check every pair of adjacent waypoints in p to examine whether obstacles appear between them. Suppose $(\mathbf{x}_i, q_B^i)$ can not reach $(\mathbf{x}_j, q_B^j)$ due to new obstacles. Then, we find the first waypoint after $(\mathbf{x}_j, q_B^j)$ (including it) that is not replaced by obstacles, denoted by $(\mathbf{x}_k, q_B^k)$. This gives rise to a subtask $((\mathbf{x}_i, q_B^i), (\mathbf{x}_k, q_B^k))$, which may include smaller subtasks. Next, we build a smaller and subtask-specific Büchi automaton B' which consists of states in the path from q_B^i to q_B^k in B. Given B', the sampling-based method in [14] is used to build one tree to find a path in the new environment for this subtask. Then, we replace the original path segment corresponding to $((\mathbf{x}_i, q_B^i), (\mathbf{x}_k, q_B^k))$ with the new path. Finally, from $(\mathbf{x}_k, q_B^k)$, we repeat until the end of p.

We conduct simulations for $m = 1, 2, 3$ obstacles. For every given m, we generate 10 different environments randomly and run 10 trials for each environment. The original paths are the paths found in Section VI-A. The results are displayed in Tab. II. We see that the runtime of our method is not significantly affected by the complexity of the task and is consistently lower than the runtime of the method in [14].

TABLE II: Runtimes in the slightly changed environment

tasks	$m = 1$		$m = 2$		$m = 3$	
	t	t [14]	t	t [14]	t	t [14]
ϕ_1	0.31	0.33	0.35	0.31	0.40	0.31
ϕ_2	0.30	0.22	0.34	0.18	0.35	0.22
ϕ_3	0.35	0.76	0.39	0.89	0.43	0.84
ϕ_4	0.38	1.10	0.34	1.18	0.42	0.98
ϕ_5	0.31	3.32	0.37	4.17	0.41	3.77
ϕ_6	0.34	9.49	0.38	10.17	0.44	13.38

VII. Conclusion

In this paper we proposed an optimal control synthesis algorithm for LTL tasks that exploits experience from solving similar LTL tasks before. Simulation showed that our method generally outperforms existing control synthesis methods in terms of time to generate feasible plans. Our algorithm is probabilistically complete and asymptotically optimal.

References

[1] S. L. Smith, J. Tumova, C. Belta, and D. Rus, "Optimal path planning for surveillance with temporal-logic constraints," *The International Journal of Robotics Research*, vol. 30, no. 14, pp. 1695–1708, 2011.

[2] Y. Chen, X. C. Ding, A. Stefanescu, and C. Belta, "Formal approach to the deployment of distributed robotic teams," *IEEE Transactions on Robotics*, vol. 28, no. 1, pp. 158–171, 2012.

[3] Y. Kantaros and M. M. Zavlanos, "Intermittent connectivity control in mobile robot networks," in *2015 49th Asilomar Conference on Signals, Systems and Computers*. IEEE, 2015, pp. 1125–1129.

[4] C. I. Vasile and C. Belta, "Sampling-based temporal logic path planning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo, Japan, November 2013, pp. 4817–4822.

[5] Y. Kantaros and M. M. Zavlanos, "Sampling-based control synthesis for multi-robot systems under global temporal specifications," in *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2017, pp. 3–14.

[6] ——, "Sampling-based optimal control synthesis for multi-robot systems under global temporal tasks," *IEEE Transactions on Automatic Control*, 2018.

[7] E. M. Wolff, U. Topcu, and R. M. Murray, "Optimization-based trajectory generation with linear temporal logic specifications," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 5319–5325.

[8] Y. Shoukry, P. Nuzzo, A. Balkan, I. Saha, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada, "Linear temporal logic motion planning for teams of underactuated robots using satisfiability modulo convex programming," in *Decision and Control (CDC), 2017 IEEE 56th Annual Conference on*. IEEE, 2017, pp. 1132–1137.

[9] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press Cambridge, 2008.

[10] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[11] K. Erol, J. A. Hendler, and D. S. Nau, "Umcp: A sound and complete procedure for hierarchical task-network planning." in *AIPS*, vol. 94, 1994, pp. 249–254.

[12] F. Bacchus and F. Kabanza, "Using temporal logics to express search control knowledge for planning," *Artificial intelligence*, vol. 116, no. 1-2, pp. 123–191, 2000.

[13] R. T. Icarte, T. Klassen, R. Valenzano, and S. McIlraith, "Using reward machines for high-level task specification and decomposition in reinforcement learning," in *International Conference on Machine Learning*, 2018, pp. 2112–2121.

[14] Y. Kantaros and M. M. Zavlanos, "Temporal logic optimal control for large-scale multi-robot systems: 10 400 states and beyond," in *2018 IEEE Conference on Decision and Control (CDC)*. IEEE, 2018, pp. 2519–2524.