Revising Motion Planning under Linear Temporal Logic Specifications in Partially Known Workspaces

Meng Guo, Karl H. Johansson and Dimos V. Dimarogonas

***Abstract*— In this paper we propose a generic framework for real-time motion planning based on model-checking and revision. The task specification is given as a Linear Temporal Logic formula over a finite abstraction of the robot motion. A preliminary motion plan is first generated based on the initial knowledge of the system model. Then real-time information obtained during the runtime is used to update the system model, verify and further revise the motion plan. The implementation and revision of the motion plan are performed in real-time. This framework can be applied to partially-known workspaces and workspaces with large uncertainties. Computer simulations are presented to demonstrate the efficiency of the framework.**

I. Introduction

Temporal-logic-based motion planning provides a fully automated correct-by-design controller synthesis approach for autonomous robots. Temporal logics such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) provide formal high level languages that can describe planning objectives more complex than the well-studied point-to-point navigation [19], [22], [24]. The task specification is given as an LTL formula with respect to a discretized abstraction of the robot motion [3], [18]. Then a high-level discrete plan is found by off-the-shelf model-checking algorithms given the finite transition system and the task specification [1], [17], [10]. This plan is then implemented through the corresponding low-level hybrid controller.

The correctness of the above framework relies on the critical assumption that the workspace is perfectly known and is correctly represented in the finite transition system. Once a motion plan is generated off-the-shelf, the robot executes the motion plan no matter what changes in the workspace, i.e., it does not react to real-time observations, as discussed in [3], [9], [14]. Consequently, this framework is not robust to model uncertainties in both the workspace and robot dynamics. When a complete representation of the workspace is not available, more sophisticated techniques are needed. In [9], [30], the robot's motion and the uncertain workspace are modeled as nondeterministic Markov decision processes, where the goal is to find a control strategy that maximizes the probability that the specification is satisfied. In [21], [31], a two-player GR(1) game between the robot and the environment is constructed and a receding horizon planning is introduced. A winning strategy for the robot can be synthesized by exhaustively searching though all possible combinations of the robot movements and the admissible workspaces. [23] updates the motion plan in real-time by "patching" it locally, where GR(1) formulas are assumed. The allowed changes are that some cells in the workspace become *unreachable*, which is only an aspect of the real-time information we introduce in Section III.A.

In this paper, instead of aiming for a motion plan off-line that takes into account every possible situation, we first create a preliminary plan based on the initially available knowledge about the robot and the workspace. Then while implementing the plan, real-time information about the system is gathered, based on which the current motion plan is verified and revised. The topic of motion plan revision is also addressed in [11] and [16], which emphasize on how to revise the task specification when this task is not realizable by the current system model. We, from an opposite viewpoint, are interested in utilizing real-time information to update the system model, then verify and revise the motion plan, while keeping the specification unchanged.

The main contribution of this work is that we propose a novel framework for real-time motion planning based on model checking and revision. We first modify the existing nested-DFS algorithm to search for an accepting path of a directed graph, which gives a preliminary motion plan. Then we classify three types of real-time information that might be obtained during real-time execution, and show how they can be used to update the system model. We provide a criterion to verify whether the current motion plan remains valid for the updated system. If not, an iterative revision algorithm is designed to revise the plan locally such that it becomes valid and fulfills the task specification. This framework is particularly useful for operation in partially-known workspaces and workspaces with uncertainties.

The rest of the paper is organized as follows: Section II briefly introduces the automaton-based model-checking framework. In Section III, we discuss about the real-time information that might be gathered during the runtime, and how it can be used to update the system model, and to verify and revise the current motion plan. Simulation examples are presented in Section IV to illustrate the proposed framework.

II. Preliminary Model-Checking

A. Task Specification in LTL

Since a task specification is normally stated as the desired behaviors of the robot within a specific workspace, the workspace we consider here is geometrically partitioned into N regions, denoted by the set $\Pi = \{\pi_0, \pi_1, \ldots, \pi_N\}$. Some

The authors are with the ACCESS Linnaeus Center, School of Electrical Engineering, KTH Royal Institute of Technology, SE-100 44, Stockholm, Sweden. mengg, kallej, dimos@kth.se. This work was supported by the Swedish Research Council (VR), the Knut and Alice Wallenberg Foundation, NoE HYCON2 and EU STREP RECONFIG. The third author is also affiliated with the KTH Centre for Autonomous Systems and is supported by VR through contract 2009-3948.

are regions of interest that are worth investigation or surveillance. Some are regions to be avoided like obstacle-occupied and forbidden areas. These regions can be in different shapes, such as triangles [3], polygons [8] and hexagons [26]. There are different cell decomposition schemes available, depending on the robot dynamics and associated control approaches. The latter will be discussed further in Section II-D.

In this paper we focus on the task specifications φ given as Linear Temporal Logic (LTL) formulas, due to their powerful expressiveness. The basic ingredients of an LTL formula are a set of atomic propositions AP and several boolean and temporal operators. LTL formulas are formed according to the following grammar [1]: $\varphi ::= \texttt{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \cup \varphi_2$, where $a \in AP$ and $\bigcirc$ (*next*), $\cup$ (*until*). For brevity, we omit the derivations of other useful operators like $\Box$ (*always*), $\Diamond$ (*eventually*), $\Rightarrow$ (*implication*) and refer the interested readers to Chapter 5 of [1] for details. AP usually reflects the properties of interest about the system. LTL is particularly useful to generate complex behaviors given some general instructions and rules of the mission. The same task specification can be stated in different ways, e.g., TASK1: "visit π_1, π_2, π_3 infinitely often and avoid all possible regions that are occupied by obstacles" is equivalent to "visit π_1, π_2, π_3 infinitely often and avoid $\pi_4, \ldots, \pi_{10}$" if we know regions $\pi_4, \ldots, \pi_{10}$ are occupied by obstacles. But the corresponding sets of AP are different. The LTL formula of the former specification is given by

$$\varphi = (\Box\Diamond a_1) \wedge (\Box\Diamond a_2) \wedge (\Box\Diamond a_3) \wedge (\Box\neg a_4), \tag{1}$$

where $\Box$, $\Diamond$ are *always* and *eventually* operators. The four APs $a_{1,2,3,4}$ are:

- $a_{1,2,3} = \{\texttt{the robot is in region } \pi_{1,2,3}\}$,
- $a_4 = \{\texttt{the robot is in a region occupied by an obstacle}\}$,

while ten atomic propositions are needed in total for the second case. This means that the way we define φ will significantly effect the size of φ and the complexity of the following procedures.

Given an LTL formula φ over AP, there is a union of infinite words that satisfy φ: $\texttt{Words}(\varphi) = \{\sigma \in (2^{AP})^w \mid \sigma \models \varphi\}$, where $\models \subseteq (2^{AP})^w \times \varphi$ is the satisfaction relation. There exists a Nondeterministic Büchi automaton (NBA) $\mathcal{A}_\varphi$ over 2^{AP} that $\texttt{Words}(\varphi) = \mathcal{L}_w(\mathcal{A}_\varphi)$, where $\mathcal{L}_w(\mathcal{A}_\varphi)$ is the language of $\mathcal{A}_\varphi$. $\mathcal{A}_\varphi$ can be constructed by first generating a generalized NBA $\mathcal{G}_\varphi$ following the proof of Theorem 5.37 in [1], which is then transformed into an equivalent NBA $\mathcal{A}_\varphi$. This process can be done in time and space $2^{\mathcal{O}(|\varphi|)}$ [1]. There are fast translation algorithms [12] from LTL to Büchi automaton, which are also available online [25]. Specifically, the NBA corresponding to φ is defined as: $\mathcal{A}_\varphi = (Q, 2^{AP}, \delta, Q_0, \mathcal{F})$, where Q is a finite set of states, Q_0 is the initial state, 2^{AP} is the input alphabet, $\delta : Q \times 2^{AP} \to 2^Q$ is a transition relation and $\mathcal{F} \subseteq Q$ is a set of accepting states. For instance, the Büchi automaton corresponding to (1) is illustrated in Figure 1 by [25].

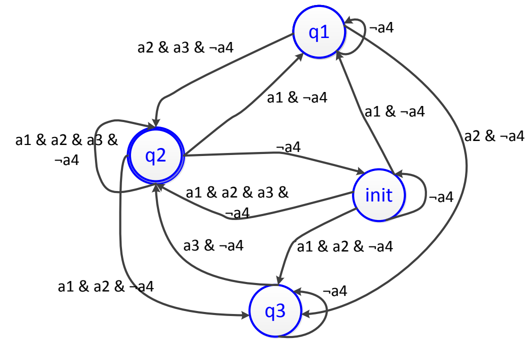

Fig. 1. Büchi automaton corresponding to (1) in TASK1. The transition from state q_1 to q_3 is given by $q_3 \in \delta(q_1, l)$ where the input alphabet $l = (a_2 \ \& \ \neg a_4)$, which is a short notation for four input alphabets $\{a_2\}$, $\{a_2, a_1\}$, $\{a_2, a_3\}$, $\{a_2, a_1, a_3\}$.

B. Discretized Abstraction

As stated in the introduction, a finite discretized abstraction [8] of the low level continuous robot motion is constructed first. Here a labeled finite transition system [1] is used as a model to describe the behavior of the robot. The partitioned workspace consists of N regions $\Pi = \{\pi_0, \pi_1, \ldots, \pi_N\}$ and the robot motion is abstracted as transitions among these regions. With a slight abuse of notation, we denote the state $\pi_i = \{\texttt{the robot is at region } \pi_i\}$. The states reflect which region the robot is currently visiting. The transitions or state changes represent that the robot has moved from one region to another. Note that the transition relation is not necessarily the adjacency relation in the geometrical sense. Instead it is defined in the control-driven fashion:

Definition 1 (Control-driven transition): There is a transition from state π_i to state π_j if an admissible controller u_{π_i, π_j} exists that could drive the robot from any point in region π_i to a point in region π_j.

The above definition is closely related to the implementation of a motion plan in Section II-D and the notion of equivalence discussed later. There are many available control techniques that drive a robot from any point inside a region to an adjacent region through a desired facet, like vector-field-based [7], navigation-function-based [19] for linear [2] or nonholonomic dynamic models [24]. Formally the control-driven finite transition system (FTS) is defined below:

Definition 2 (Control-driven FTS): The control-driven FTS is a tuple $\mathcal{T}_c = (\Pi, \to_c, \Pi_0, AP, L_c)$, where

- $\Pi = \{\pi_i\}$, $\pi_i = \{\texttt{the robot is in region } \pi_i\}$, $i = 1, \cdots, N$.
- $\to_c \subseteq \Pi \times \Pi$ is the transition relation by Definition 1
- $\Pi_0 \subseteq \Pi$ is the set of initial states.
- AP is the set of atomic propositions specified by φ.
- $L_c(\pi_i) : \Pi \to 2^{AP}$ is a labeling function.

We omit the Act element for simplicity. The labeling function $L_c(\pi_i)$ is the subset of AP which are true at state π_i. By Definition 7.1 in [1], the finite transition system $\mathcal{T}_c$ and the underlying controlled continuous system are bisimulation-equivalent, i.e., following a sequence of transitions in $\mathcal{T}_c$ is equivalent to applying the controllers paired with each transition in the continuous system. $\mathcal{T}_c$ needs both the knowledge about the robot dynamics and the workspace property. So if the workspace is only partially-known, the constructed $\mathcal{T}_c$ might not reflect the actual system. In this case, it is called the preliminary finite transition system.

We assume that $\mathcal{T}_c$ does not have a terminal state [1]. The successors of π_i are defined as $\texttt{Post}(\pi_i) = \{\pi_k \in \Pi \,|\, \pi_i \rightarrow_c \pi_k\}$ while predecessors of π_i are $\texttt{Pre}(\pi_i) = \{\pi_k \in \Pi \,|\, \pi_k \rightarrow_c \pi_i\}$. An infinite path fragment p of $\mathcal{T}_c$ is an infinite state sequence $p = \pi_0\pi_1\pi_2\ldots$ such that $\pi_i \in \texttt{Post}(\pi_{i-1})$ for all $i > 0$. Its trace is the sequence of atomic propositions that are true in the states along the path, i.e., $\texttt{trace}(p) = L_c(\pi_0)L_c(\pi_1)L_c(\pi_2)\cdots$. The trace of $\mathcal{T}_c$ is defined as $\texttt{Trace}(\mathcal{T}_c) = \cup_{p\in I}\texttt{trace}(p)$, where I is the set of all infinite paths in $\mathcal{T}_c$. The satisfaction relation $p \models \varphi$ if and only if $\texttt{trace}(p) \in \texttt{Words}(\varphi)$, where φ is an LTL formula over the same AP. We intend to find an infinite path p of $\mathcal{T}_c$ that satisfies φ, where p is called a motion plan.

C. Product Automaton

There are several well-established methods to find an infinite path p of $\mathcal{T}_c$ that $p \models \varphi$ as in [1], [6] and [29]. In this paper, we use the automaton-based model-checking approach, see [29] and Algorithm 11 in [1]. It is based on checking the emptiness of the product Büchi automaton. Since $\texttt{Words}(\varphi) = \mathcal{L}_w(\mathcal{A}_\varphi)$ and $\texttt{trace}(p) \in \texttt{Trace}(\mathcal{T}_c)$, the original problem is equivalent to finding the intersection $\texttt{Trace}(\mathcal{T}_c) \cap \mathcal{L}_w(\mathcal{A}_\varphi)$, which is actually the language of $\mathcal{T}_c \otimes \mathcal{A}_\varphi$. This product automaton $\mathcal{A}_p = \mathcal{T}_c \otimes \mathcal{A}_\varphi$ accepts all runs which are valid for $\mathcal{T}_c$ and at the same time satisfies φ.

Note here that unlike Algorithm 11 in [1], we do not negate the task specification φ when applying the model-checking algorithm. This is because we are interested in the "good" behavior of the system that satisfies the specification, not the "bad" behavior that satisfies the negated specification for the purpose of verification. Formally, $\mathcal{A}_p$ is defined as:

Definition 3 (Product automaton): The product Büchi automaton [1] $\mathcal{A}_p = \mathcal{T}_c \otimes \mathcal{A}_\varphi = (Q',\, 2^{AP},\, \delta',\, Q'_0,\, \mathcal{F}')$, where

- $Q' = \Pi \times Q$. $q' = \langle \pi,\, q\rangle \in Q'$, $\forall \pi \in \Pi$ and $\forall q \in Q$.
- $\langle \pi_j,\, q_n\rangle \in \delta'(\langle \pi_i,\, q_m\rangle)$ iff $\pi_j \in \texttt{Post}(\pi_i)$ and $q_n \in \delta(q_m,\, L_c(\pi_j))$.
- $Q'_0 = \{\langle \pi,\, q\rangle \,|\, \pi \in \Pi_0,\, q \in Q_0\}$, set of initial states.
- $\mathcal{F}' = \{\langle \pi,\, q\rangle \,|\, \pi \in \Pi,\, q \in \mathcal{F}\}$, set of accepting states.

Algorithm 1 computes $\mathcal{A}_p$ given $\mathcal{T}_c$ and $\mathcal{A}_\varphi$. An accepting run [1] of $\mathcal{A}_p$ consists of two parts: a part that is executed only once from an initial state to one accepting state and the second part which is repeated infinitely from an accepting state back to itself. It can be represented by a finite accepting path $\mathcal{P}$ of the directed state graph $G(\mathcal{A}_p)$ of $\mathcal{A}_p$. As shown in Figure 2, an accepting path $\mathcal{P}$ of $G(\mathcal{A}_p)$ always has the following structure [5]:

$$\mathcal{P} = \underbrace{q'_0\, q'_1\, q'_2 \cdots q'_k}_{\mathcal{P}_l}\,(\underbrace{q'_f \cdots\cdots q'_f}_{\mathcal{P}_c})^\omega\,, \tag{2}$$

where $q'_0 \in Q'_0$ and $q'_f \in \mathcal{F}'$. $\mathcal{P}_l$ is a finite sequence of states from an initial state q'_0 to an accepting state q'_f while $\mathcal{P}_c$ is a cycle from q'_f to q'_f. Since the size of $G(\mathcal{A}_p)$ is finite, there are finite number of states and edges in $\mathcal{P}$. Similar as in [1], we define $\texttt{suffix}(x, \mathcal{P})$ in Algorithm 3 and 5 as the segment of $\mathcal{P}$ after state x, not including x; if x appears in $\mathcal{P}$ several times, we choose the last x. Similarly, $\texttt{suffix}((x,\, y), \mathcal{P})$ in Algorithm 5 is defined as the segment of $\mathcal{P}$ after edge $(x,\, y)$, including y; if $(x,\, y)$ appears in $\mathcal{P}$ several times, we choose the first $(x,\, y)$. Finally, $\texttt{prefix}((x,\, y), \mathcal{P})$ in Algorithm 5 is defined as the segment of $\mathcal{P}$ before edge $(x,\, y)$, not including x; if $(x,\, y)$ appears in $\mathcal{P}$ several times, we choose the first $(x,\, y)$. For any finite sequence of states $\mathcal{P}$, we denote $\texttt{node}(\mathcal{P}) = \{\, q'_s \,|\, q'_s \in Q',\, q'_s \text{ appears in } \mathcal{P} \text{ at least once}\,\}$ and $\texttt{edge}(\mathcal{P}) = \{(q'_s,\, q'_h) \,|\, q'_h \in \delta'(q'_s),\; (q'_s,\, q'_h) \text{ appears in } \mathcal{P} \text{ at least once}\}$.

We modify the nested-DFS algorithm of [5], [13] to favor Algorithm 5, where function $\texttt{M-DFS}$ is reused. In particular, compared with "traditional" DFS that returns a feasible path from an initial state to a goal state, Algorithm 2 takes a directed graph G, one state q, and a set of states Q in G as inputs, and returns a feasible path from q to one state belonging to Q, if one exists. The main Algorithm 3 takes a directed graph G and a set of initial states Q_0, and returns one accepting path of G. In Algorithm 3, $stack1$ stands for $\mathcal{P}_l$ while $stack2$ stands for $\mathcal{P}_c$ in (2). Note that if the inner search at line 10 returns an empty $stack2$, the outer search is resumed to search for the next reachable accepting state. With an accepting path $\mathcal{P}$ of $G(\mathcal{A}_p)$ in hand, a preliminary motion plan, namely an infinite path p in $\mathcal{T}_c$ can be generated as a projection of $\mathcal{P}$ onto the state space Π of $\mathcal{T}_c$. Its trace should then automatically satisfy φ as proved in [29]. It can be verified that *worst-case* time complexity of Algorithm 3 is in $\mathcal{O}(|Q_0| \cdot |G|^2 \cdot |\mathcal{F}|)$, where $|G|$ is the size of G.

Algorithm 1: Function $\texttt{Product(}\,\mathcal{T}_c,\, \mathcal{A}_\varphi)$, by Definition 3

Input: FTS $\mathcal{T}_c$, NBA $\mathcal{A}_\varphi$ over the same AP
Output: the product automaton $\mathcal{A}_p$
foreach $q'_h = \langle \pi_i,\, q_m\rangle \in Q'$ **do**
  if $q_m \in Q_0$ *and* $\pi_i \in \Pi_0$ **then**
    $q'_h \in Q'_0$;
  if $q_m \in \mathcal{F}$ **then**
    $q'_h \in \mathcal{F}'$;
  foreach $q'_s = \langle \pi_j,\, q_n\rangle \in Q'$ **do**
    if $\pi_j \in \texttt{Post}(\pi_i)$ *and* $q_n \in \delta(q_m,\, L_c(\pi_j))$ **then**
      $q'_s \in \delta'(q'_h)$;

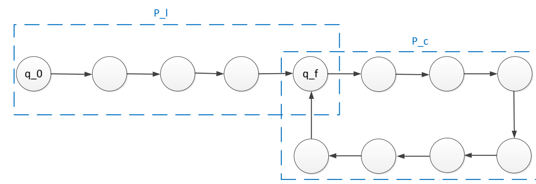

Fig. 2. Two parts of an accepting path

Algorithm 2: Function M-DFS(G, q, Q), modified depth-first search algorithm

```
Input: directed graph G, state q ∈ G, a set of states
       Q ⊆ G
Output: a path of G from q to one state in Q
stack = (q) ;
expstack = ∅ ;
while stack ≠ ∅ do
    x = top(stack);
    if x is adjacent to a vertex y ∉ expstack then
        add y to the top of stack and expstack ;
        if y ∈ Q then
            return stack ;
    else
        remove x from the stack ;
```

Algorithm 3: Function N-DFS(G, Q_0), modified nested-DFS algorithm [6]

```
Input: directed graph G, set of initial states Q0
Output: an accepting path P = [stack1, stack2]
F={the set of accepting states in G} ;
foreach q0 ∈ Q0 do
    stack1 = (q0) ;
    expstack = ∅ ;
    while stack1 ≠ ∅ do
        x = top(stack1) ;
        if x is adjacent to a vertex y ∉ expstack then
            add y to the top of stack1 and expstack ;
            if y ∈ F then
                stack2 = M-DFS(G, y, stack1) ;
                if stack2 ≠ ∅ then
                    add suffix(top(stack2), stack1)
                    to the top of stack2 ;
                    return stack1, stack2 ;
        else
            remove x from the stack1 ;
P = [stack1 stack2]
```

D. Continuous Controller Synthesis

A preliminary motion plan essentially represents an infinite sequence of regions to visit. Definition 1 states that there exists an admissible controller paired with each transition in $\mathcal{T}_c$. Then a lower-level hybrid controller [3] that implements the motion plan is synthesized by combining those continuous controllers. As mentioned in [3], [9], [14], this hybrid control scheme is normally built off-line and fixed once it is synthesized. If $\mathcal{T}_c$ is built based on perfect knowledge of the workspace and the robot dynamics, then this motion plan is provably correct and can be implemented directly. However if $\mathcal{T}_c$ is not aligned with the real workspace due to model uncertainty or limited priori knowledge, the implementation and revision of the motion plan need to be performed in real time.

III. Real-time Verifying and Revision

The autonomous robot is required to operate in a partially known workspace while being capable of gathering real-time information about the workspace, which is then used to verify and revise the motion plan.

A. Real-time Information

For simplicity, we assume that any information is gathered when the robot reaches a region, not during the transition from one region to another. We define three different types of real-time information that may be obtained at time $t > 0$:

- type A information: $(\pi_i, \pi_j) \in A(t)$ if π_j is added to $\texttt{Post}(\pi_i)$. $(\pi_i, \pi_j) \in A_{\neg}(t)$ if π_j is deleted from $\texttt{Post}(\pi_i)$.
- type B information: denote $(b, \pi_i) \in B(t)$ if the labeling function of state π_i is updated to $b \subseteq 2^{AP}$, i.e., $L_c(\pi_i) = b$.
- type C information: represents dynamic behaviors of the environment.

For brevity, denote by $Info(t) = \{A_{\neg}(t), A(t), B(t)\}$, the set of information obtained at time $t > 0$. The information source can be either from on-board sensing measurements or communication with external stations. Note that type C information is included here for completeness. It will not be discussed in this paper, and is a topic of ongoing research [4], [21], [28]. We assume the workspace is partially-known and static. Type A information about (π_i, π_j) will be updated at most once. Regarding type B information about π_i, any element belonging to $L_c(\pi_i)$ can be changed at most once. Namely, the transition relations and properties of the regions are fixed once being identified.

B. Update $\mathcal{A}_p$ and Verification

Given the type A and B information, the first question is how to utilize these information to update $\mathcal{A}_p$ from Section II-C. We use the notation $\mathcal{A}_p^-$ and $\mathcal{A}_p^+$ to distinguish the product automaton before and after an update. In particular, the updating rules from $\mathcal{A}_p^-$ to $\mathcal{A}_p^+$ are:

Definition 4 (Updating Rules): $\mathcal{A}_p^-$ is updated to $\mathcal{A}_p^+$ by $Info(t)$ at time t following the rules below:

1) If $(\pi_i, \pi_j) \in A(t)$, $\langle \pi_j, q_n \rangle$ is added to $\delta'(\langle \pi_i, q_m \rangle)$, $\forall\, q_n, q_m$ satisfying $q_n \in \delta(q_m, L_c(\pi_j))$.
2) If $(\pi_i, \pi_j) \in A_{\neg}(t)$, $\langle \pi_j, q_n \rangle$ is deleted from $\delta'(\langle \pi_i, q_m \rangle)$, $\forall q_m, q_n \in Q$.
3) If $(b, \pi_j) \in B(t)$, then $\forall \pi_i \in \texttt{Pre}(\pi_j)$:
   - $\langle \pi_j, q_n \rangle$ is added to $\delta'(\langle \pi_i, q_m \rangle), \forall\, q_n \in \delta(q_m, b)$,
   - $\langle \pi_j, q_n \rangle$ is deleted from $\delta'(\langle \pi_i, q_m \rangle)$, $\forall\, q_n \notin \delta(q_m, b)$.

The set of removed edges at t is denoted by $R(t) \subset Q' \times Q'$.

Remark 1: Another possible approach would be to update the finite transition system $\mathcal{T}_c$ first. Then $\mathcal{A}_p^+$ can be re-constructed by Definition 3 with the updated $\mathcal{T}_c$. We chose to revise $\mathcal{A}_p^-$ directly because the size of $\mathcal{A}_p$ is *exponential* in the length of φ, meaning that it would be computationally

Algorithm 4: Function `Update(` $\mathcal{A}_p^-$, $Info(t)$), by Definition 4

Input: current $\mathcal{A}_p^-$, update information $Info(t)$
Output: updated $\mathcal{A}_p^+$, the set of removed edges $R(t)$
$\{A_\neg(t),\, A(t),\, B(t)\} = Info(t)$;
foreach $(\pi_i,\, \pi_j) \in A(t)$ **do**
  foreach $q_n \in \delta(q_m,\, L_c(\pi_j))$ **do**
    $q'_h = \langle \pi_i,\, q_m\rangle$ and $q'_s = \langle \pi_j,\, q_n\rangle$;
    add q'_s to $\delta'(q'_h)$;
foreach $(\pi_i,\, \pi_j) \in A_\neg(t)$ **do**
  foreach $q_n \in \delta(q_m,\, L_c(\pi_j))$ **do**
    $q'_h = \langle \pi_i,\, q_m\rangle$ and $q'_s = \langle \pi_j,\, q_n\rangle$;
    remove q'_s from $\delta'(q'_h)$;
    add $(q'_h,\, q'_s)$ to $R(t)$;
foreach $(b,\, \pi_j) \in B(t)$ **do**
  foreach $\pi_i \in Pre(\pi_j)$ **do**
    foreach $q_n \in \delta(q_m,\, b)$ **do**
      $q'_h = \langle \pi_i,\, q_m\rangle$ and $q'_s = \langle \pi_j,\, q_n\rangle$;
      add q'_s to $\delta'(q'_h)$;
    foreach $q_n \notin \delta(q_m,\, b)$ **do**
      $q'_h = \langle \pi_i,\, q_m\rangle$ and $q'_s = \langle \pi_j,\, q_n\rangle$;
      remove q'_s from $\delta'(q'_h)$;
      add $(q'_h,\, q'_s)$ to $R(t)$;

expensive to construct the product automaton from scratch each time a new piece of information is obtained.

Definition 4 is summarized in Algorithm 4. Note that the corresponding state graph $G(\mathcal{A}_p)$ is also updated. Suppose $\mathcal{P}^-$ is an accepting path of $\mathcal{A}_p^-$. In case new information is obtained, $\mathcal{A}_p^-$ is updated to $\mathcal{A}_p^+$ by Definition 4. Two natural questions arise: 1. is $\mathcal{P}^-$ an accepting path of $G(\mathcal{A}_p^+)$? 2. if not, how can we modify $\mathcal{P}^-$ such that it satisfies the accepting condition of $G(\mathcal{A}_p^+)$. The first question is answered in this part and the second is addressed in Section III-C.

Assumption 1: $(\pi_i,\, \pi_j) \in A_\neg(t)$ is obtained before the first time that the transition from π_i to π_j is taken. $\pi_i \in B(t)$ is obtained before the first time that π_i is reached.

Theorem 1: Assume $\mathcal{P}^-$ is an accepting path of $G(\mathcal{A}_p^-)$. $\mathcal{A}_p^-$ is updated to $\mathcal{A}_p^+$ based on $Info(t)$. Then $\mathcal{P}^-$ remains to be an accepting path of $G(\mathcal{A}_p^+)$ if and only if $R(t)$ and the set of edges in $\mathcal{P}^-$ have no common elements, i.e., $R(t) \cap \mathtt{edge}(\mathcal{P}^-) = \emptyset$.

Proof: For the necessity part, if $R(t) \cap \mathtt{edge}(\mathcal{P}^-) = \emptyset$, it means that $\mathcal{P}^-$ remains a valid path of $G(\mathcal{A}_p^+)$. Since $\mathcal{P}^-$ is an accepting path of $G(\mathcal{A}_p^-)$, it contains a path from an initial state to an accepting state and a cycle back to this accepting state. Moreover, since $\mathcal{A}_p^-$ and $\mathcal{A}_p^+$ have the same set of initial states and accepting states, $\mathcal{P}^-$ is an accepting path of $G(\mathcal{A}_p^+)$. For the sufficiency, if $R(t) \cap \mathtt{edge}(\mathcal{P}^-) \neq \emptyset$, it means that at least one edge in $\mathcal{P}^-$ is invalid thus $\mathcal{P}^-$ is not a valid path of $G(\mathcal{A}_p^+)$, and thus it is also not an accepting path. ■

Lemma 2: Assume $\mathcal{P}_0$ is an accepting path of $G(\mathcal{A}_p^0)$,

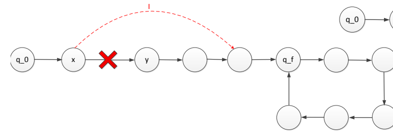

Fig. 3. Path revision, case I.

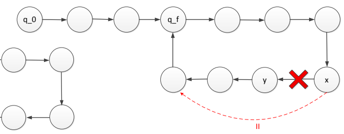

Fig. 4. Path revision, case II

where $\mathcal{A}_p^0$ is the preliminary product automaton at $t = 0$. Then $\mathcal{P}_0$ remains valid through the runtime if $R(t) \cap \mathtt{edge}(\mathcal{P}_0) = \emptyset$, $\forall t > 0$.

Proof: The proof follows by applying Theorem 1 iteratively. ■

Lemma 2 can be thought as "lucky" cases where a preliminary motion plan may be valid during the whole runtime even though the system is updated frequently. Namely, the part of the product automaton relevant to the preliminary plan is correct initially. For example, a preliminary plan for TASK1 may be feasible and obstacle free even though large number of obstacles may exist in the workspace. However, since the motion plan generated by model checking is not necessarily unique, it is very likely that the preliminary plan is refuted during the runtime.

C. Real-time Revision

Theorem 1 states that if $R(t) \cap \mathtt{edge}(\mathcal{P}^-) \neq \emptyset$, then $\mathcal{P}^-$ is not an accepting path of $G(\mathcal{A}_p^+)$. The next question is how to deal with the falsified $\mathcal{P}^-$. One possible solution could be to recall Algorithm 3 with respect to the updated graph $G(\mathcal{A}_p^+)$, which returns an accepting path $\mathcal{P}^+$ of $G(\mathcal{A}_p^+)$. But this method requires a complete nested-DFS of $G(\mathcal{A}_p^+)$, which is computationally heavy (the worst-case time complexity $\mathcal{O}(2^{|\varphi|} \cdot |\Pi|)$ from [1]) in case of large $\mathcal{T}_c$ and complex φ, especially when $\mathcal{A}_p$ has to be updated frequently.

Instead we are interested in revising the current path $\mathcal{P}^-$ locally such that it fulfills the accepting condition of $G(\mathcal{A}_p^+)$. The idea is that since most of the path segments belonging to $\mathcal{P}^-$ are still valid for $G(\mathcal{A}_p^+)$, we only need to make up the edges that are not valid in $G(\mathcal{A}_p^+)$. The framework is provided in Algorithm 5 and summarized below. Let $(x,\, y) \in R(t) \cap \mathtt{edge}(\mathcal{P}^-)$. Depending on the location of the removed edge $(x,\, y)$ in an accepting path (2), there are three possibilities: first, $(x,\, y) \in stack1$ but $(x,\, y) \notin stack2$. As shown in Figure 3, we want to find a "bridge" path $bridge1$ that connects state x to the suffix part of $stack1$ after $(x,\, y)$. Function `M-DFS` in Algorithm 2 serves this purpose. If such a bridging path $bridge1$ exists, $stack1$ is revised by inserting $bridge1$ between the prefix part of $stack1$ before $(x,\, y)$ and the suffix part of $stack1$ that $bridge1$ is connected to, as in Line 8 of Algorithm 5. Secondly, if $(x,\, y) \in stack2$ but $(x,\, y) \notin stack1$, a similar procedure can be applied to $stack2$, as in Figure 4. $stack2$ is revised by inserting $bridge2$ between the prefix part of $stack2$ before $(x,\, y)$ and the suffix part of $stack2$ that $bridge2$ is connected to, as in Line 14 of Algorithm 5. Thirdly, it is also possible that $(x,\, y) \in stack1$ and $(x,\, y) \in stack2$. The previous two procedures are applied and $stack1$ and $stack2$ are both revised. Last but not least, if any of the above calls to function `M-DFS` returns an empty $bridge1$ or $bridge2$, it means that the current accepting state is not reachable from x. Then Algorithm 3 is applied to $G(\mathcal{A}_p^+)$, to find an accepting path from x to another accepting state and cycles back. Note that $\mathcal{P}^-$ is revised iteratively and the condition $R(t) \cap \texttt{edge}(\mathcal{P}^-) \neq \emptyset$ is also checked iteratively.

Theorem 3: An accepting path of $G(\mathcal{A}_p^+)$ can always be found by Algorithm 5 if one exists.

Proof: Since $bridge1$, $bridge2$, $nstack1$ and $nstack2$ are derived from $G(\mathcal{A}_p^+)$, they are always valid for $G(\mathcal{A}_p^+)$. Whenever $stack1$ and $stack2$ are revised at Line 8, 11, 12, 16, 19, 20, the number of elements within $R(t) \cap \texttt{edge}(\mathcal{P}^-)$ is decreased at least by one. Since the size of $R(t)$ is finite, the "while" loop in Algorithm 5 will eventually terminate and return a valid accepting path $\mathcal{P}^+$ of $G(\mathcal{A}_p^+)$, i.e., $\mathcal{P}^-$ needs to be revised at most as many times as the size of $R(t)$. This completes the proof. ∎

To conclude, Algorithm 6 provides the complete structure of the real-time motion planner based on real-time model checking and revision. Given the preliminary transition system and the task specification, a product automaton is constructed based on Definition 1, an accepting path of which is obtained by Algorithm 3. This accepting path is implemented by the corresponding hybrid controller. Whenever a piece of new information is perceived during the runtime, the product automaton is firstly updated by Algorithm 4 and then the current motion plan is revised by Algorithm 5 if it contains any removed edges. Implementation and revision of the motion plan are performed in real-time.

D. On-the-fly Construction and Applicability

As discussed in [1] and [5], an interesting aspect of the automaton-based model-checking algorithm is that it can be executed on-the-fly, meaning that $\mathcal{T}_c$, $\mathcal{A}_\varphi$ and the product automaton $\mathcal{A}_p$ can be generated in parallel with searching for an accepting path of $\mathcal{A}_p$. In our case, a complete construction of $\mathcal{A}_p$ by Algorithm 1 can also be avoided. The adjacency relation needed at Line 7 of Algorithm 3 and Line 5 of Algorithm 2 can be built "on demand". Namely the transition relation from q'_h to q'_s is verified only when q'_h is visited. That is to say, when Algorithm 3 terminates and returns an accepting path, only relevant parts of the entire product automaton $\mathcal{A}_p$ are constructed. Moreover, it is worth mentioning that our framework can be integrated with other path searching methods concerning optimality [15], [28]. A weighted finite transition system [28] could be introduced to take into account the cost of each transition and minimal cost path search algorithms from [15], [22] can be used in Algorithm 2, 3 and 5 to replace the depth-first search.

Our framework can be applied to various workspaces without modifying φ since the task specifications are given as general operation rules and full knowledge of the workspace is not required. For instance, (1) can be applied to any partitioned workspace consisting of three regions of interests, independent of its size or inner structure.

Algorithm 5: Function `Revise`($G(\mathcal{A}_p^+)$, $\mathcal{P}^-$, $R(t)$), revision of the current path

Input: the accepting path $\mathcal{P}^-$ before update, directed graph $G(\mathcal{A}_p^+)$, R from Algorithm 4
Output: an accepting path $\mathcal{P}^+$ of $G(\mathcal{A}_p^+)$
[$stack1$, $stack2$] = $\mathcal{P}^-$;
while $(R(t) \cap \textit{edge}(\mathcal{P}^-)) \neq \emptyset$ **do**
  $(x,\, y) \in (R(t) \cap \texttt{edge}(\mathcal{P}^-))$;
  initialize $bridge1$ and $bridge2$ as nonempty;
  if $(x,\, y) \in stack1$ **then**
    $bridge1$=`M-DFS`($G(\mathcal{A}_p^+)$, x, `suffix`($(x,\, y)$, $stack1$)) ;
    if $bridge1 \neq \emptyset$ **then**
      $stack1$=[`prefix`($(x,\, y)$, $stack1$) $bridge1$ `suffix`(`top`($bridge1$), $stack1$)];
    else
      [$nstack1$, $nstack2$]=`N-DFS`($G(\mathcal{A}_p^+)$, x) ;
      $stack1$=[`prefix`($(x,\, y)$, $stack1$) $nstack1$] ;
      $stack2 = nstack2$;
  if $(x,\, y) \in stack2$ **then**
    $bridge2$=`M-DFS`($G(\mathcal{A}_p^+)$, x, `suffix`($(x,\, y)$, $stack2$)) ;
    if $bridge2 \neq \emptyset$ **then**
      $stack2$=[`prefix`($(x,\, y)$, $stack2$) $bridge2$ `suffix`(`top`($bridge2$), $stack2$)];
    else
      [$nstack1$, $nstack2$]=`N-DFS`($G(\mathcal{A}_p^+)$, x) ;
      $stack1$=[$stack1$ `prefix`($(x,\, y)$, $stack2$) $nstack1$] ;
      $stack2 = nstack2$;
  $\mathcal{P}^- = [stack1\, stack2]$;
$\mathcal{P}^+ = \mathcal{P}^-$

IV. Example — Surveillance Robot

Consider a unicycle robot that satisfies: $\dot{x_0} = v\cos\theta$, $\dot{y_0} = v\sin\theta$, $\dot{\theta} = w$, where $\mathbf{p}_0 = (x_0,\, y_0)^T \in \mathbb{R}^2$ is the center of mass, $\theta \in [0,\, 2\pi]$ is the orientation, and $v, w \in \mathbb{R}$ are the transition and rotation velocities. The robot is equipped with on-board sensors that gather real-time information about the workspace while operating. This workspace consists of three regions of interest and several regions are occupied by obstacles. The surveillance task is given by "visit region A, B, C infinitely often and avoid all possible obstacles". As discussed in Section II-A, four atomic propositions are needed and the corresponding LTL task specification is given by (1). Its associated NBA is shown in Figure 1. All simulations are carried out in MATLAB on a desktop computer (3.06 GHz Duo CPU and 8GB of RAM). All computations were accomplished within one second.

A. Example one

In the first example, we consider the unicycle model and the 6×6 workspace in Figure 5, where each cell has

Algorithm 6: Function RM-MC($\mathcal{T}_c^0$, $\mathcal{A}_\varphi$, $Info(t)$), real-time model checker

Input: preliminary $\mathcal{T}_c^0$, NBA $\mathcal{A}_\varphi$, real-time information
Output: accepting path of the current $\mathcal{A}_p$
$\mathcal{A}_p^-$ = `Product`($\mathcal{T}_c^0$, $\mathcal{A}_\varphi$) ;
Q_0'={the set of initial states in $\mathcal{A}_p^-$} ;
$\mathcal{P}^-$ = `N-DFS`($G(\mathcal{A}_p^-)$, Q_0');
while *implementing the plan* $\mathcal{P}^-$ **do**
  if $Info(t) \neq \emptyset$ **then**
    [$\mathcal{A}_p^+$, $R(t)$] = `Update`($\mathcal{A}_p^-$, $Info(t)$) ;
    $\mathcal{P}^+$= `Revise`($G(\mathcal{A}_p^+)$, $\mathcal{P}^-$, $R(t)$) ;
    $\mathcal{P}^- = \mathcal{P}^+$ and $\mathcal{A}_p^- = \mathcal{A}_p^+$

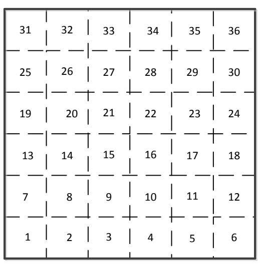

Fig. 5. Initially known workspace

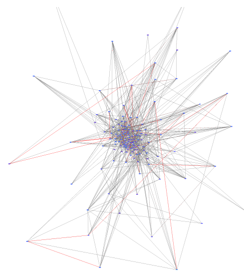

Fig. 6. Preliminary product automaton

the size $r_s \times r_s$. Regions A, B, C are regions 6, 31, 36 respectively. There exist continuous controllers that steer the robot from any cell to one of its four geometrically neighboring cells. Specifically, it has four motion primitives "`up, down, left, right`", where the "`up`" motion can be implemented by letting $v = v_{\text{const}}$, $w = \frac{r_s}{2\, v_{\text{const}}}$, until $\theta = \frac{\pi}{2}$, then $v = v_{\text{const}}$, $w = 0$ for time period $t = \frac{r_s}{v_{\text{const}}}$. The other three primitives can be defined similarly.

By Definition 2, the preliminary finite transition system is given by $\mathcal{T}_c = (\Pi, \rightarrow_c, \Pi_0, AP, L_c)$, where $\Pi = \{1, 2, \cdots 36\}$, $\Pi_0 = \{1\}$, $AP = \{a_1, a_2, a_3, a_4\}$ as defined in Section II-A, and L_c is the labeling function indicating if the robot is at regions 6, 31, 36, or in collision with the obstacles. The workspace is initialized as obstacle free as in Figure 5. It is not known a priori how many obstacles there are and how they are deployed in the workspace. The actual workspace is shown in Figure 8, where regions in gray are occupied by obstacles and there are walls (black lines) between some neighboring regions.

The framework in Algorithm 6 is applied here. The preliminary $\mathcal{A}_p^0$ as shown in Figure 6 has $36 \times 4 = 144$ states and is constructed following Definition 3. One of its accepting paths $\mathcal{P}_0$ is obtained by Algorithm 3 (red line in Figure 6), from which a preliminary motion plan is built. For simplicity, we assume that the robot receives new information twice during the runtime (once for the lower half workspace and once for the upper half workspace). In practice, when and how much information is obtained rely heavily on the sensing ability of the robot. The robot is capable of gathering

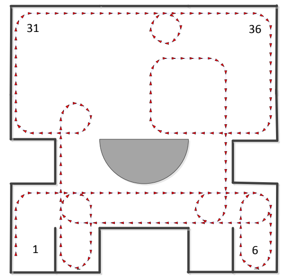

Fig. 7. Motion plan after the first update

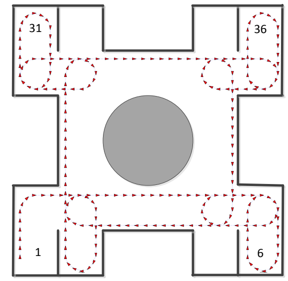

Fig. 8. Motion plan after the second update

two kinds of information: 1. if there is a wall between the current region and its neighboring regions; 2. which of the nearby regions are occupied by obstacles. Clearly the first one belongs to type A information and the second is type B information in Section III-A.

At region 1, the robot obtains the information: $A_\neg = \{(1, 2), (2, 1), (5, 6), (6, 5)\}$ and $B = \{(a_4, 13), (a_4, 15), (a_4, 3), (a_4, 4), (a_4, 16), (a_4, 18)\}$, meaning that going directly from region 1 to region 2 and vice versa are not allowed due to the presence of a wall in the middle (same for regions 5 and 6), and regions 13, 15, 3, 4, 16, 18 are detected to be occupied by obstacles. Then the product automaton is updated accordingly by Algorithm 4, where 324 edges are removed in total. Then Algorithm 5 is used to revise the current accepting path recursively, where function `M-DFS` is called only 10 times. The updated motion plan is illustrated by the arrowed red lines in Figure 7, which is valid for the currently-known workspace, but not valid for the actual workspace. The robot follows this motion plan until it reaches region 14. At region 14, the robot obtains the information: $A_\neg = \{(31, 32), (32, 31), (35, 36), (36, 35)\}$ and $B = \{(a_4, 33), (a_4, 34), (a_4, 21), (a_4, 22), (a_4, 19), (a_4, 24)\}$ and the procedure is repeated. Function `M-DFS` is called 13 times before an accepting path is found. The updated plan is valid for the actual workspace and fulfills the task specification, as illustrated in Figure 8.

B. Example two

In our second example, we consider again the unicycle model and the workspace in Figure 9, which consists of 12 polygonal regions. Regions A, B, C in φ are regions 2, 3, 4 respectively. The continuous controller that drives the robot from an region to any geometrically adjacent region is based on [24], which is built by constructing vector fields over each cell for each face. The controller design is not stated here for brevity. The preliminary workspace is initialized as obstacles free and the corresponding $\mathcal{T}_c$ is constructed as before. The actual workspace is shown in Figure 10, where region 9 is occupied by obstacles and there are walls between some regions. The robot is capable of perceiving the same types of information as described in the previous example. The product automaton has $12 \times 4 = 48$ states. A preliminary motion plan is generated by Algorithm 3 (arrowed red line

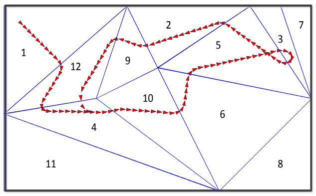

Fig. 9. Initial workspace and the preliminary motion plan

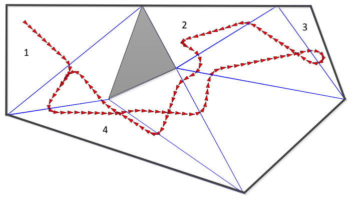

Fig. 10. Actual workspace and the final motion plan

in Figure 9), but it is not valid for the actual workspace as it intersects with region 9, which is occupied by obstacles.

The robot moves according to the motion plan and reaches region 12, where it obtains the following information: $A_\neg = \{(4, 11), (11, 4)\}$, and $B = \{(a_4, 9)\}$. The product automaton is updated accordingly, where 71 edges are removed in total. The revision is finished after function `M-DFS` is called only two times. The updated motion plan is illustrated by the arrowed red lines in Figure 10. Then the robot follows this updated motion plan. At region 6 and 3, it obtains the information: $A_\neg = \{(6, 8), (8, 6)\}$ and $A_\neg = \{(7, 3), (3, 7)\}$, respectively. 32 edges are removed in both cases. But the motion plan remains valid because its corresponding accepting path does not contain any of the removed edges. The final trajectory is shown in Figure 10.

V. Conclusion

In this paper we proposed a generic framework for real-time motion planning based on iterative model checking and revision. This framework is particularly useful for partially known workspace and workspace with large uncertainties. Real-time sensory information is used to update the system model, based on which the motion plan is revised locally and iteratively. Future work could include dynamic environments and multiple robots.

References

[1] C. Baier, J.-P Katoen. Principles of model checking. *The MIT Press*, 2008.

[2] C. Belta, V. Isler, G. J. Pappas. Discrete abstractions for robot motion planning and control in polygonal environments. *IEEE Transactions on Robotics*, 21(5): 864-874, 2005.

[3] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, G. J. Pappas. Symbolic planning and control of robot motion. *IEEE Robotics and Automation Magazine*, 14: 61-71, 2007.

[4] Y. Chen, J. Tumova, C. Belta. LTL Robot motion control based on automata learning of environmental dynamics. *IEEE International Conference on Robotics and Automation (ICRA)*, 2012.

[5] C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Computer-aided Verification*, 1992.

[6] E. M. Clarke, O. Grumberg, D. A. Peled. Model checking. *The MIT Press*, 1999.

[7] J. Desai, J. Ostrowski, V. Kumar. Controlling formations of multiple mobile robots. *IEEE Int. Conf. Robotics and Automation*, 2864-2869, 1998.

[8] X. Ding, M. Kloetzer, Y. Chen, C. Belta. Automatic deployment of robotic teams. *IEEE Robotics Automation Magazine*, 18: 75-86, 2011.

[9] X. C. Ding, S. L. Smith, C. Belta, D. Rus. LTL control in uncertain environments with probabilistic satisfaction guarantees. *18th IFAC World Congress*, 2011.

[10] G. E. Fainekos, A. Girard, H. Kress-Gazit, G. J. Pappas. Temporal Logic Motion Planning for Dynamic Mobile Robots. *Automatica*, 45(2): 343-352, 2009.

[11] G. E. Fainekos. Revising temporal logic specifications for motion planning. *IEEE Conference on Robotics and Automation*, 2011.

[12] P. Gastin, D. Oddoux. Fast LTL to Büchi automaton translation. *In Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, 2001.

[13] G. J. Holzmann, D. Peled, M. Yannakakis. On nested depth first search. *American Mathematical Society*, 1997.

[14] S. Karaman, E. Frazzoli. Sampling-based algorithms for optimal motion planning. *International Journal of Robotics Research*, 30(7): 846-894, 2011.

[15] S. Karaman, E. Frazzoli. Vehicle routing with linear temporal logic specifications: Applications to Multi-UAV Mission Planning. *Navigation, and Control Conference in AIAA Guidance*, 2008.

[16] K. Kim, G. E. Fainekos, S. Sankaranarayanan. On the revision problem of specification automaton. *IEEE International Conference on Robotics and Automation*, 5171-5176, 2012.

[17] M. Kloetzer, C. Belta, A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Transactions on Automatic Control*, 53(1): 287-297, 2008.

[18] M. Kloetzer, C. Belta. Automatic deployment of distributed teams of robots from temporal logic specifications. *IEEE Transactions on Robotics*, 26(1): 48-61, 2010.

[19] D. E. Koditschek, E. Rimon. Robot navigation functions on manifolds with boundary. *Advances Appl. Math.*, 11:412-442, 1990.

[20] H. Kress-Gazit, T. Wongpiromsarn, U. Topcu. Correct, reactive robot control from abstraction and temporal logic specifications. *IEEE Robotics and Automation Magazine*, 2011.

[21] H. Kress-Gazit, G. E. Fainekos, G. J. Pappas. Temporal Logic-based Reactive Mission and Motion Planning. *IEEE Transactions on Robotics*, 25(6): 1370-1381, 2009.

[22] S. M. LaValle. Planning algorithms. *Cambridge University Press*, 2006.

[23] S. C. Livingston, R. M. Murray, J. W. Burdick. Backtracking temporal logic synthesis for uncertain environments. *In IEEE International Conference on Robotics and Automation*, 51635170, 2012.

[24] S. R. Lindemann, I. I. Hussein, S. M. LaValle. Real time feedback control for nonholonomic mobile robots with obstacles. *45th IEEE Conference on Decision and Control*, 2406-2411, 2006.

[25] D. Oddoux, P. Gastin. LTL2BA software: fast translation from LTL formulae to Büchi automaton. `http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/index.php`.

[26] A. S. Oikonomopoulos, S. G. Loizou, K. J. Kyriakopoulos. Coordination of multiple non-holonomic agents with input constraints. *Proceedings of the IEEE International Conference on Robotics and Automation*, 869-874, 2009.

[27] N. Piterman, A. Pnueli, Y. Saar. Synthesis of reactive(1) designs. *Proc. VMCAI*, 364-380, 2006.

[28] A. Ulusoy, S. L. Smith, X. Ding, C. Belta. Robust multi-robot optimal path planning with temporal logic constraints. *IEEE International Conference on Robotics and Automation (ICRA)*, 2012.

[29] M. Y. Vardi, P. Wolper. An automaton-theoretic approach to automatic program verification. *IEEE Computer Society*, 332-344, 1986.

[30] E. M. Wolff, U. Topcu, R. M. Murray. Robust Control of Uncertain Markov Decision Processes with Temporal Logic Specifications. *American Control Conference (ACC)*, 2012.

[31] T. Wongpiromsarn, U. Topcu, R. M. Murray. Receding Horizon Temporal Logic Planning for Dynamical Systems. *IEEE Conference on Decision and Control*, 2009