Article

The International Journal of Robotics Research
2014, Vol. 33(12) 1593–1607
© The Author(s) 2014
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/0278364914537008
ijr.sagepub.com
SAGE

Receding horizon temporal logic control in dynamic environments

Alphan Ulusoy and Calin Belta

Abstract

We present a receding horizon method for controlling an autonomous vehicle that must satisfy a rich mission specification over service requests occurring at the regions of a partitioned environment. The overall mission specification consists of a temporal logic statement over a set of static, a priori known requests, a regular expression over a set of dynamic requests that can be sensed only locally, and a servicing priority order over these dynamic requests. Our approach is based on two main steps. First, we construct an abstraction for the motion of the vehicle in the environment by using input–output linearization and assignment of vector fields to the regions in the partition. Second, a receding horizon controller computes local plans within the sensing range of the vehicle such that both local and global mission specifications are satisfied. We implement and evaluate our method through experiments and simulations consisting of a quadrotor performing a persistent surveillance task over a planar grid environment.

Keywords

Receding horizon control, temporal logic, formal methods

1. Introduction

Temporal logics have been traditionally used to specify the correctness of computer programs (Baier and Katoen, 2008). They recently gained popularity in robotics due to their ability to express complex robotics tasks (Kress-Gazit et al., 2007; Bhatia et al., 2010; Wongpiromsarn et al., 2010; Ding et al., 2011; Kress-Gazit et al., 2011; Karaman and Frazzoli, 2011). Given a high-level mission specification expressed as a temporal logic formula over the properties satisfied at the states of a finite motion model, tools from formal verification and automata games (Baier and Katoen, 2008) can be used to automatically generate motion plans and control strategies.

The goal of this paper is to automatically control a robot operating in a dynamic environment such that it correctly reacts to the events sensed locally, while at the same time performing some other higher-level task. Consider, for example, a persistent surveillance mission in a post-disaster setting, where an autonomous flying vehicle is required to keep on photographing known damage areas. Simultaneously, the vehicle is also required to look for survivors, fires, and gas leakages, which can be sensed locally around the vehicle using onboard sensors. Even though the works cited above consider synthesis of provably correct, and sometimes optimal robot behavior, there are very few works in the literature that tackle the control synthesis problem when locally sensed dynamical events are part of the mission specification. Kress-Gazit et al. (2007) and Wongpiromsarn et al. (2010) present methods that use a fragment of linear temporal logic (LTL) called general reactivity (1) (GR(1)) as a specification language. While this temporal logic can capture dynamic events and the proposed synthesis algorithms can synthesize reactive control policies, the synthesis algorithms themselves are not reactive, i.e. they do not become simpler in the absence of some events. Since these algorithms have to take all dynamic events into account, the resulting massive state space significantly hinders their scalability. Ding et al. (2012) consider a control problem on a finite graph, where the aim is to maximize locally collected rewards while satisfying a mission specification given as an LTL formula over some properties satisfied at the vertices of the graph. As in this paper, a receding horizon approach, motivated by the local sensing of rewards, is shown to guarantee the satisfaction of the LTL specification in infinite time. However, the algorithms proposed by Ding et al. (2012) are limited to maximizing locally collected rewards and cannot be trivially modified to obtain rich local behaviors that are of our interest.

Division of Systems Engineering, Boston University, Boston, MA, USA

Corresponding author:
Alphan Ulusoy, Division of Systems Engineering, Boston University, 15 St Mary's Street, Brookline, MA 02446, USA.
Email: alphan@bu.edu

Maly et al. (2013) propose an iterative multi-layer framework to solve the motion planning problem for a robot that must satisfy a given temporal logic specification in a partially unknown environment. Even though their approach can efficiently replan for locally discovered environmental constraints such as obstacles, they do not consider the case where locally discovered dynamical events are also part of the mission specification. Given a temporal logic specification that cannot be satisfied under some cost constraint, Kim and Fainekos (2013) propose a method to revise the specification such that the revised specification can be satisfied under the given cost constraint and is as close as possible to the original specification. However, they consider a purely static scenario without any dynamical events. Moreover, Kress-Gazit et al. (2007), Ding et al. (2012), Maly et al. (2013) and Kim and Fainekos (2013) consider a partition of the entire environment and compute their high-level plans over this partition. Thus, their methods scale with the size of the overall environment. In Ulusoy et al. (2013b), we considered an environment where properties satisfied at the regions changed non-deterministically, however we did not consider dynamic requests occurring at arbitrary locations.

The main contribution of this work is an efficient control synthesis algorithm for vehicles operating in dynamic environments. Inspired by the above post-disaster scenario, we consider mission specifications consisting of two parts: a *global* specification given as a temporal logic formula over a set of *static* requests occurring at known locations of a known map (e.g. surveying the known damage areas), and a *local* specification given as a regular expression and a servicing priority order over *dynamic* requests sensed locally (e.g. avoid unsafe regions, pick up and drop off locally detected items of type 1 and 2, pick items of type 1 first if both types of items are detected). Our approach can be summarized as follows. Initially, we map the global specification to an automaton that guides the vehicle such that it satisfies the global specification in the absence of locally sensed events. During deployment, according to events sensed locally, a local automaton is generated and linked to the global automaton in such a way that the satisfaction of the global specification is still guaranteed. In order to ensure timely responses to dynamically changing events, the control strategy is implemented in a receding horizon fashion. The high-level, automata theoretic control strategies are refined into vehicle controllers by using input–output linearization, polytopic partitions of the environment, and vector field assignments based on polytope-to-polytope controllers (Belta and Habets, 2006). Thus, in contrast with the methods proposed by Kress-Gazit et al. (2007), Wongpiromsarn et al. (2010), Ding et al. (2012), Maly et al. (2013) and Kim and Fainekos (2013) our approach initially considers only static requests and plans for dynamic requests only within the local sensing range and only as required. Consequently, the complexity of our approach scales only with the number of static requests and the local sensing range of the robot as opposed to the size of the environment. Furthermore, as opposed to the approach given by Ding et al. (2012), we consider continuous vehicle dynamics and richer local specifications that allow both ordering among dynamic requests as well as avoiding them. Another contribution is the successful implementation of this computational framework in a simulator and in an experimental setup involving an autonomous quadrotor flying in an indoor environment equipped with a motion capture system.

A preliminary version of our approach appeared in the proceedings of Robotics: Science and Systems (RSS) 2013 (Ulusoy et al., 2013a). However, in Ulusoy et al. (2013a) the local specification is limited to a servicing priority order over the dynamic requests. While such local specifications can express the order in which the dynamic requests must be serviced in the case of multiple simultaneous requests, e.g. if both a survivor and a fire are detected always assist the survivor before extinguishing the fire, they cannot express the ordering between dynamic requests that occur at different times, e.g. a dropoff cannot occur before a pickup. Here, we extend this preliminary work by considering a much richer class of local specifications consisting of a regular expression that determines the servicing order among dynamic requests that occur at different times in addition to the servicing priority order explained above. Furthermore, the approach presented in Ulusoy et al. (2013a) requires translation of a syntactically co-safe LTL formula to an automaton, which has exponential time complexity, at each time step during online execution. The approach presented in this paper does not require such an operation. We also analyze the complexity of our approach, discuss the details of low-level controllers used in the experiments, and provide new case studies.

The rest of the paper is organized as follows. In Section 2, we give necessary definitions and preliminaries in formal methods. In Section 3, we formally state the problem that we consider in this paper, and present our solution in Section 4. We present our experimental results in Section 5. We conclude with final remarks in Section 6.

2. Preliminaries

In this section, we provide a brief review of concepts related to automata theory and formal verification and introduce our notation. We refer the interested reader to Baier and Katoen (2008) and references therein for a more detailed treatment of these topics. For a set Π, we use $|\Pi|$ and 2^{Π} to denote its cardinality and power set, respectively.

Definition 2.1 (Transition system). *A (weighted, deterministic) transition system is a tuple* $\mathbf{T} := (Q_T, q_{T,init}, \delta_T, \Pi_T, h_T, w_T)$*, where:*

- Q_T *is the finite set of states;*
- $q_{T,init} \in Q_T$ *is the initial state;*
- $\delta_T \subseteq Q_T \times Q_T$ *is the transition relation;*
- Π_T *is the finite set of atomic propositions;*

- $h_T : \mathcal{Q}_T \rightarrow 2^{\Pi_T}$ *is the labeling function giving the set of atomic propositions satisfied at a state;*
- $w_T : \delta_T \rightarrow \mathbb{N}$ *assigns a non-negative weight to each transition.*

A *run* of $\mathbf{T}$ is a sequence of states $q^0, q^1, \ldots$ such that $q^0 = q_{T,init}$, $q^k \in \mathcal{Q}_T$, and $(q^k, q^{k+1}) \in \delta_T$ for all k. A run generates a *word* $\omega^0, \omega^1, \ldots$, where $\omega^k = h_T(q^k)$ is the set of atomic propositions satisfied at state q^k.

LTL is an extension of (Boolean) propositional logic that can capture temporal relations (Baier and Katoen, 2008). LTL formulas are interpreted over infinite words generated by the transition system $\mathbf{T}$ from Definition 2.1. Informally, an LTL formula ϕ over a set of atomic propositions Π_T can involve Boolean operators $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), and temporal operators $\mathbf{G}$ (always), $\mathbf{F}$ (eventually), $\mathbf{X}$ (next), and $\mathcal{U}$ (until). For instance, $\mathbf{G}\,\mathrm{p}$ states that p is true at all positions of the word, $\mathbf{F}\,\mathrm{p}$ states that p eventually becomes true in the word, and $\mathbf{X}\,\mathrm{p}$ states that proposition p is true at the next position of the word. Formula $\mathrm{p}_1\,\mathcal{U}\,\mathrm{p}_2$ states that there exists $k \geq 0$ such that w^k satisfies p_2 and w^j satisfies p_1 for all $0 \leq j < k$, where w^k is the symbol at the kth position of the word. More expressivity can be achieved by combining the temporal and Boolean operators. We say a run of $\mathbf{T}$ satisfies ϕ if and only if the word generated by the run satisfies ϕ.

Definition 2.2 (**Büchi automaton).** *A Büchi automaton is a tuple* $\mathbf{B} := (\mathcal{Q}_B, \mathcal{Q}_{B,init}, \delta_B, \Sigma_B, \mathcal{F}_B)$*, where:*

- $\mathcal{Q}_B$ *is the finite set of states;*
- $\mathcal{Q}_{B,init} \subseteq \mathcal{Q}_B$ *is the set of initial states;*
- $\delta_B \subseteq \mathcal{Q}_B \times \Sigma_B \times \mathcal{Q}_B$ *is the (non-deterministic) transition relation;*
- Σ_B *is the input alphabet;*
- $\mathcal{F}_B \subseteq \mathcal{Q}_B$ *is the set of accepting (final) states.*

A *run* of $\mathbf{B}$ over an input word $\omega^0, \omega^1, \ldots$ is a sequence of states $q^0, q^1, \ldots$, such that $q^0 \in \mathcal{Q}_{B,init}$, and $(q^k, \omega^k, q^{k+1}) \in \delta_B$, for all k. A Büchi automaton $\mathbf{B}$ accepts a word over Σ_B if and only if at least one of the corresponding runs intersects with $\mathcal{F}_B$ infinitely many times. For any LTL formula ϕ over a set Π_T, one can construct a Büchi automaton with input alphabet $\Sigma_B = 2^{\Pi_T}$ accepting all and only words over 2^{Π_T} that satisfy ϕ using automated tools such as *ltl2ba* given in Gastin and Oddoux (2001).

A *regular expression* φ contains propositions from Π_T connected by operators such as | (union), * (Kleene star, i.e. repetition), and . (concatenation) (Baier and Katoen, 2008). For example, $\mathrm{p}_1^*.\mathrm{p}_3$ defines the set of strings where p_1 occurs zero or more times followed by p_3 and $(\mathrm{p}_1|\mathrm{p}_2)^*$ defines the set of strings where p_1 or p_2 occurs zero or more times in arbitrary order. We say a finite run of $\mathbf{T}$ satisfies φ if and only if the word generated by the run satisfies φ.

Definition 2.3 (**Finite state automaton).** *A (deterministic) finite state automaton (FSA) is a tuple* $\mathbf{A} := (\mathcal{Q}_A, q_{A,init}, \delta_A, \Sigma_A, \mathcal{F}_A)$*, where:*

- $\mathcal{Q}_A$ *is the finite set of states;*
- $q_{A,init} \in \mathcal{Q}_A$ *is the initial state;*
- $\delta_A : \mathcal{Q}_A \times \Sigma_A \times \mathcal{Q}_A$ *is the deterministic transition relation;*
- Σ_A *is the input alphabet;*
- $\mathcal{F}_A \subseteq \mathcal{Q}_A$ *is the set of accepting (final) states.*

A *run* of $\mathbf{A}$ over an input word $\omega^0, \ldots, \omega^{n-1}$ is a sequence of states $q^0, \ldots, q^n$ such that $q^0 = q_{A,init}$ and $(q^k, \omega^k, q^{k+1}) \in \delta_A$ for all k. An FSA $\mathbf{A}$ accepts a word of length n over Σ_A if and only if the corresponding run ends in some $q^n \in \mathcal{F}_A$. For any regular expression over a set Π_T, one can construct the corresponding FSA using automated tools such as *dk.brics.automaton* (Møller, 2011) or *JFLAP* (Rodger and Finley, 2006).

3. Problem formulation and approach

In this section we introduce the control synthesis problem with temporal logic constraints for a vehicle operating in a dynamic environment. For simplicity of presentation, the problem is formulated for a vehicle that can deterministically move among the adjacent cells of a grid environment. At the end of this section (Remark 3.3), we show that this is enough for a large class of problems including vehicles with non-trivial dynamics. In Section 5.2, we present the details of such an implementation for an autonomous quadrotor.

3.1. Environment and vehicle model

Consider a (planar) grid environment defined as

$$\mathcal{E} = (\mathcal{C}, \mathcal{S}, \mathcal{L}_s, \mathcal{D}, \mathcal{L}_d(t)), \tag{1}$$

where $\mathcal{C} = \{c_{x,y} \mid 0 \leq x < m, 0 \leq y < n\}$ is an $m \times n$ array of square cells, $\mathcal{S}$ is the set of *static requests* that can be serviced at the cells of the environment, $\mathcal{L}_s : \mathcal{C} \rightarrow \mathcal{S}$ is a (possibly partial) map that gives the location of the static requests on the grid, $\mathcal{D}$ is the set of *dynamic requests* that occur at arbitrary cells of the environment whose locations are not known in advance and can only be discovered (sensed) locally, and $\mathcal{L}_d(t) : \mathcal{C} \rightarrow \mathcal{D}$ is a time varying (possibly partial) map that gives the location of the dynamic requests on the grid. We assume that $c_{0,0}$ corresponds to the cell at the southwest corner of the array when looked from an East–North–Up (ENU) coordinate frame (Figure 1) and a cell cannot have both a static and a dynamic request.

In this paper, we consider the case where a vehicle navigates in the environment by either holding position or moving to one of the four cells sharing a facet with its current cell, and has the ability to sense the dynamic requests occurring in the cells within its vicinity. Specifically, the vehicle can sense an $o \times p$ array of cells where $o > 1, p > 1$, both o and p are odd. The center cell of this sensing grid corresponds to the current cell of the vehicle in the environment. After arriving at a cell, the vehicle uses its sensors to detect the dynamic requests occurring at this subset of $\mathcal{C}$, i.e. it

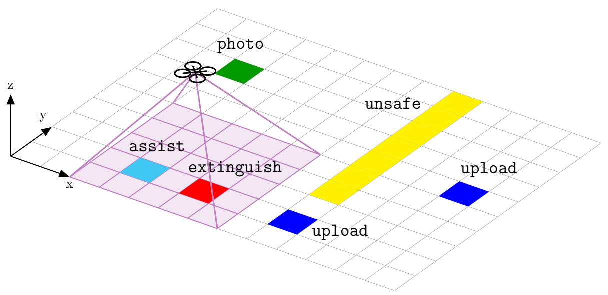


Fig. 1. A schematic representation of the scenario considered in Example 3.1. We have two static requests defined in the environment with $\mathcal{S} = \{\texttt{photo}, \texttt{upload}\}$, $\mathcal{L}_s(c_{2,7}) = \texttt{photo}$, $\mathcal{L}_s(c_{8,1}) = \texttt{upload}$, and $\mathcal{L}_s(c_{11,5}) = \texttt{upload}$. The cells within the sensing range of the quadrotor are highlighted in violet. Cells $c_{3,1}$ and $c_{5,1}$ are shown in cyan and red; and correspond to locally detected `assist` and `extinguish` dynamic requests, respectively.

discovers the portion of the time varying map $\mathcal{L}_d(t)$ corresponding to the cells that fall within its sensing range. We also assume that the vehicle has the ability to complete a static or dynamic request at a cell by visiting that cell.

Example 3.1. *Figure 1 gives a schematic representation of an* 13×10 *environment, where a quadrotor is required to perform a persistent surveillance task. The set of static requests that can be serviced at the regions of the environment is* $\mathcal{S} = \{\texttt{photo}, \texttt{upload}\}$ *and these static requests are assigned to the cells such that* $\mathcal{L}_s(c_{2,7}) = \texttt{photo}$, $\mathcal{L}_s(c_{8,1}) = \texttt{upload}$, *and* $\mathcal{L}_s(c_{11,5}) = \texttt{upload}$. *In Figure 1, these cells are highlighted in green and blue, respectively. The sensing capability of the quadrotor is modeled by a* 5×5 *grid of square cells (highlighted in violet in Figure 1) where the center cell of this grid corresponds to the area directly underneath the quadrotor. The set of dynamically occurring requests whose locations are not known a priori is* $\mathcal{D} = \{\texttt{unsafe}, \texttt{extinguish}, \texttt{assist}\}$. *Requests* `extinguish` *and* `assist` *correspond to 'extinguish fire' and 'assist survivor', respectively, whereas cells with* `unsafe` *request should be always avoided. In Figure 1, cells with locally detected dynamic requests* `extinguish` *and* `assist` *are shown in red and cyan, respectively.*

3.2. Problem formulation

In this work, we consider two types of specifications to define the overall mission of the vehicle: a global mission specification and a local mission specification. The global mission specification is an LTL formula ϕ_g (Section 2) over the set of static requests $\mathcal{S}$ and dictates the global motion of the vehicle in the environment. The local mission specification specifies how the vehicle must respond to the dynamic requests detected within the sensing range of the vehicle. Specifically, the local mission specification consists of a priority function $prio : \mathcal{D} \to \mathbb{N}$, where lower numbers correspond to higher priorities, and a regular expression ϕ_l over the set $\mathcal{D}$ of dynamic requests. The priority function specifies a total or partial order in which simultaneously detected multiple dynamic requests must be serviced, e.g. if both a survivor and a fire is detected assist the survivor, while ϕ_l selectively enables or disables the dynamic requests that can be serviced at a given time based on previously serviced dynamic requests, e.g. `dropoff1` can be serviced only after `pickup1` and once `pickup1` is serviced it cannot be serviced again until `dropoff1`. If there are dynamic requests in $\mathcal{D}$ that do not appear in ϕ_l, then the vehicle must avoid them at all times, e.g. avoid areas that are `unsafe` for flight. We can now formulate the problem that we consider in this paper.

Problem 3.2. *Given an environment* $\mathcal{E}$, *a global mission specification* ϕ_g *in the form of an LTL formula over* $\mathcal{S}$, *and a local mission specification consisting of a priority function* $prio : \mathcal{D} \to \mathbb{N}$ *and a regular expression* ϕ_l *over* $\mathcal{D}$, *find a vehicle control strategy such that the produced trajectory satisfies both the global and the local mission specifications.*

Example 3.1 Revisited. *The quadrotor given in Figure 1 is required to complete a persistent surveillance mission starting at* $c_{2,7}$. *The global mission specification is to*

> *"Keep taking photos and upload current photo before taking another photo,"*

which can be expressed in LTL as

$$\phi_g := \mathbf{GF}\,\texttt{photo} \wedge \mathbf{G}(\texttt{photo} \Rightarrow \mathbf{X}\texttt{upload}) \wedge \mathbf{G}(\texttt{upload} \Rightarrow \mathbf{X}\texttt{photo}). \quad (2)$$

The local mission specification is to

> *"Avoid unsafe areas, assist survivors, extinguish fires, and if both fire and survivor are detected assist the survivor first,"*

which translates to

$$\phi_l := (\texttt{extinguish}|\texttt{assist})^*, \quad prio(\texttt{assist}) = 0,\ prio(\texttt{extinguish}) = 1. \quad (3)$$

Note that since the dynamic request `unsafe` $\in \mathcal{D}$ *does not appear in* ϕ_l, *the vehicle must avoid it at all times.*

Our solution to Problem 3.2 takes the form of a receding horizon controller, which computes the next cell the vehicle must go to from its current cell such that the order in which the static requests are serviced satisfies the global mission specification ϕ_g and the order in which the locally sensed dynamic requests are serviced satisfies the local mission specification given by ϕ_l and *prio*. In summary, the controller that we propose works as follows. First, a global product automaton $\mathbf{G}$ that captures the motion of the vehicle between the cells with static requests and ϕ_g is constructed (Section 4.1). Then, the FSA $\mathbf{L}$ that corresponds to ϕ_l is constructed. Each time the vehicle arrives at a cell, a local transition system **U** that captures the local information obtained from the sensors and the motion of the vehicle between the cells is constructed (Section 4.2). Next, our controller uses **G**, **L**, and **U** to compute the next cell the vehicle must go such that both the global and the local mission specifications are satisfied (Section 4.2). Finally, a low-level controller moves the vehicle to this target cell. As a demonstration of our approach, we consider the case where the vehicle is a quadrotor as given in Example 3.1. In our experiments, we generate vector fields that guarantee smooth trajectories between the current and the next grid cell and use feedback linearization to stabilize the quadrotor along these trajectories (Section 5.2).

Remark 3.3. *There are two apparently restrictive assumptions in the above problem formulation. First, it is assumed that the vehicle can stay inside a cell and can move from one cell to an adjacent desired cell without penetrating to another neighbor cell. Note that these control problems can be easily solved for vehicles such as unicycles and quadrotors by input–output linearization and construction of a vector field enforcing the desired motion of a reference point among the cells. Thus, polytopic partitioning of the workspace does not require the robot to have linear dynamics (see Kloetzer and Belta (2006) for unicycles and Section 5.2 for a short discussion of such an implementation in quadrotors). Second, the partition does not have to be rectangular. This approach works for arbitrary polytopic partitions by means of additional triangulations (Belta et al., 2005). Note that non-polytopic regions resulting from other types of partitioning schemes can, in principle, be under- or over-approximated by polytopic regions.*

4. Problem solution

The controller that we propose as a solution to Problem 3.2 is presented in the form of Algorithm 1. In the following, we discuss each step of Algorithm 1 in detail.

4.1. Offline computation

Lines 1–6 of Algorithm 1 are responsible for the *offline* initialization of our algorithm and are executed only once. The first step of our solution (line 1 of Algorithm 1) is to construct the global product automaton **G** as the product of the transition system **T** that models the behavior of the vehicle between the cells with static requests and the Büchi automaton **B** that captures the global mission specification ϕ_g. To this end, we first construct a transition system $\mathbf{T} := (\mathcal{Q}_{\mathrm{T}}, q_{\mathrm{T},init}, \delta_{\mathrm{T}}, \Pi_{\mathrm{T}}, h_{\mathrm{T}}, w_{\mathrm{T}})$ (Definition 2.1) representation of the motion of the vehicle between the cells with static requests in the environment $\mathcal{E}$ given in (1). Let $c_{init} \in \mathcal{C}$ denote the initial cell of the vehicle in the environment and $\mathcal{C}_s$ denote the set of cells with static requests, i.e. $\mathcal{L}_s(q)$ is defined for all $q \in \mathcal{C}_s$. We define the set of states of **T** as $\mathcal{Q}_{\mathrm{T}} = \mathcal{C}_s \cup \{c_{init}\}$ such that we have a state for each

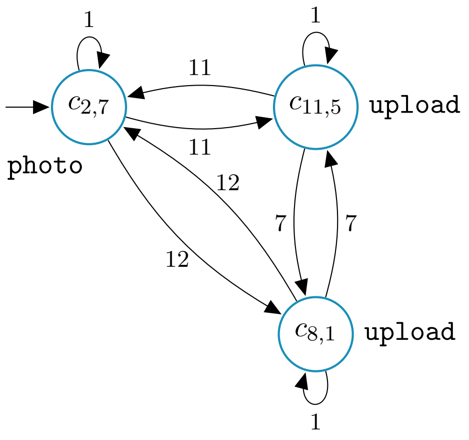


Fig. 2. Transition system **T** modeling the motion of the quadrotor among the cells with static requests for the environment given in Figure 1. Static requests are shown next to their corresponding states and $q_{\mathrm{T},init} = c_{2,7}$. Weights of the edges between two different cells give the length of the shortest path between those cells that does not go through any other cell in $\mathcal{Q}_{\mathrm{T}}$. Weights of the self-loops are set to 1 to capture the fact that the online portion of our algorithm runs periodically at each time step.

cell with a static request and the initial cell of the vehicle if $c_{init} \notin \mathcal{C}_s$. Then, we define $\Pi_{\mathrm{T}} = \mathcal{S}$, set $h_{\mathrm{T}}(q) = \mathcal{L}_s(q)$ for all $q \in \mathcal{C}_s$ and also set $h_{\mathrm{T}}(c_{init}) = \emptyset$ if $c_{init} \notin \mathcal{C}_s$. Next, we define the transitions (edges) between each pair of states with static requests, i.e. $(q, q') \in \delta_{\mathrm{T}}$ if $q, q' \in \mathcal{C}_s$. If $c_{init} \notin \mathcal{C}_s$, then we also define outgoing transitions from c_{init} to all other states in $\mathcal{C}_s$. The weight $w_{\mathrm{T}}(q, q')$ of the transition between $q, q' \in \mathcal{Q}_{\mathrm{T}}$ is the time it takes to travel the shortest path between the corresponding cells that does not go through any other cell in $\mathcal{Q}_{\mathrm{T}}$. Due to the particular implementation of the continuous controller that drives the vehicle from a cell to one of its four neighbor cells (see Section 5.2), we use Manhattan distance to calculate w_{T}. Note that each state $q \in \mathcal{C}_s$ has a self-loop so that the vehicle can stay at a given cell (corresponding continuous controllers are described in Section 5.2). Interpreting the time it takes for the vehicle to travel a single cell as unit time, we set the weights of these loops to 1 so that the movement of the vehicle in the environment is not blocked due to zero weight self-loops. Setting the weights of these self-loops to 1 also captures the fact that the vehicle performs sensing and planning at uniform intervals, i.e. at each time step, regardless of whether it is traveling or holding position.

Example 3.1 Revisited. *Figure 2 illustrates the transition system* **T** *modeling the motion of the quadrotor between the cells with static requests* photo *and* upload *for the environment given in Figure 1.*

Next, we obtain the Büchi automaton **B** that corresponds to the global mission specification ϕ_g (Gastin and Oddoux, 2001) and construct the product automaton $\mathbf{G} := \mathbf{T} \otimes \mathbf{B}$ as defined next.

Algorithm 1: Vehicle Controller.

```
Input: Environment $\mathcal{E}$, global mission specification $\phi_g$, local mission specification $prio : \mathcal{D} \to \mathbb{N}$ and $\phi_l$.
Offline Initialization (Section 4.1):
1  Construct the global product automaton $\mathbf{G} = \mathbf{T} \otimes \mathbf{B}$ (Definition 4.1).
2  Remove all of those states in $\mathcal{F}_G$ that cannot reach themselves.
3  if $\mathcal{F}_G = \emptyset$ then Abort: $\phi_g$ cannot be satisfied.
4  Let $fd(q) = \min_{q' \in \mathcal{F}_G} shortest\_dist(q, q')$ for all $q \in \mathcal{Q}_G$.
5  $g_{cur} = \arg\min_{q \in \mathcal{Q}_{G,init}} fd(q)$.
6  Construct the FSA L corresponding to $\phi_l$ and set $l_{cur} = q_{L,init}$.
Online Receding Horizon Control (Section 4.2):
7  while True do
8      Construct the local transition system U using Algorithm 2.
9      $d^\star = \infty$, $cell^\star_{next}$ = None, $g^\star_{next}$ = None.
10     $\mathcal{D}_{service} = \{h_U(q) \mid q \in \mathcal{Q}_U \wedge \exists l_{next} \in \mathcal{Q}_L \text{ s.t. } (l_{cur}, h_U(q), l_{next}) \in \delta_L\}$.
11     $\mathcal{Q}_s = \{q \mid q \in \mathcal{Q}_U \wedge \mathcal{L}_s(q) \text{ is defined}\}$, $\mathcal{Q}_d = \{q \mid q \in \mathcal{Q}_U \wedge h_U(q) \neq \emptyset\}$.
12     if $\mathcal{D}_{service} = \emptyset$ then
13         foreach $g_{next} \in \{g_{next} \mid (g_{cur}, g_{next}) \in \delta_G\}$ do
14             if $g_{next}[0] \in \mathcal{Q}_U$ then
15                 $\mathcal{Q}_{avoid} = (\mathcal{Q}_s \cup \mathcal{Q}_d) \setminus \{g_{next}[0]\}$.
16                 $\mathcal{Q}_{target} = \{g_{next}[0]\}$.
17             else
18                 $\mathcal{Q}_{avoid} = \mathcal{Q}_s \cup \mathcal{Q}_d$.
19                 $\mathcal{Q}_{target}$ = {Boundary cells of U} $\setminus \mathcal{Q}_{avoid}$.
20             $paths = shortest\_path(q_{U,init}, \mathcal{Q}_{avoid})$.
21             foreach $q \in \mathcal{Q}_{target}$ do
22                 $d_{plan} = len(paths[q]) + man\_dist(q, g_{next}[0]) + fd(g_{next})$.
23                 if $d_{plan} < d^\star$ then $d^\star = d_{plan}$, $g^\star_{next} = g_{next}$, $cell^\star_{next} = paths[q][1]$.
24     else
25         $\mathcal{Q}_{target} = \{q \mid h_U(q) \in \mathcal{D}_{service} \wedge prio(h_U(q)) = \min_{p \in \mathcal{D}_{service}} prio(p)\}$.
26         $\mathcal{Q}_{avoid} = (\mathcal{Q}_s \cup \mathcal{Q}_d) \setminus \mathcal{Q}_{target}$.
27         $paths = shortest\_path(q_{U,init}, \mathcal{Q}_{avoid})$.
28         foreach $q \in \mathcal{Q}_{target}$ do
29             if $len(paths[q]) < d^\star$ then $d^\star = len(paths[q])$, $cell^\star_{next} = paths[q][1]$.
30     if $d^\star = \infty$ then Abort: No feasible local plan.
31     if $cell^\star_{next} = g^\star_{next}[0]$ then $g_{cur} = g^\star_{next}$.
32     if $h_U(cell^\star_{next}) \neq \emptyset$ then $l_{cur} = l_{next}$ s.t. $(l_{cur}, h_U(cell^\star_{next}), l_{next}) \in \delta_L$.
33     Apply controls to reach $\text{cell}^\star_{\text{next}}$.
```

Definition 4.1 (Product of T and B). *The product of a weighted transition system* $\mathbf{T} := (\mathcal{Q}_T, q_{T,init}, \delta_T, \Pi_T, h_T, w_T)$ *(Definition 2.1) and a Büchi automaton* $\mathbf{B} := (\mathcal{Q}_B, \mathcal{Q}_{B,init}, \delta_B, \Sigma_B, \mathcal{F}_B)$ *(Definition 2.2) is a tuple* $\mathbf{G} := (\mathcal{Q}_G, \mathcal{Q}_{G,init}, \delta_G, w_G, \mathcal{F}_G)$*, where:*

- $\mathcal{Q}_G \subseteq \mathcal{Q}_T \times \mathcal{Q}_B$ *is the finite set of states that are reachable from* $\mathcal{Q}_{G,init}$*;*
- $\mathcal{Q}_{G,init} = \{(q_{T,init}, q'_B) \mid (q_B, h_T(q_{T,init}), q'_B) \in \delta_B,\ q_B \in \mathcal{Q}_{B,init}\}$ *is the set of initial states;*
- $\delta_G = \{((q_T, q_B), (q'_T, q'_B)) \mid (q_T, q'_T) \in \delta_T,\ (q_B, h_T(q'_T), q'_B) \in \delta_B\}$ *is the transition relation;*
- $w_G((q_T, q_B), (q'_T, q'_B)) = w_T(q_T, q'_T)$ *for all* $((q_T, q_B), (q'_T, q'_B)) \in \delta_G$ *is the weight function;*
- $\mathcal{F}_G = \{(q_T, q_B) \mid (q_T, q_B) \in \mathcal{Q}_G, q_B \in \mathcal{F}_B\}$ *is the set of accepting states.*

Note that the global product automaton **G** captures both the motion of the vehicle between the cells with static requests and the global mission specification, hence the name *global* product automaton.

Since the acceptance condition for a Büchi automaton is to visit an accepting state infinitely often (Section 2), in line 2 of Algorithm 1 we remove all those accepting states of **G** that cannot reach themselves. In line 3 of Algorithm 1, we

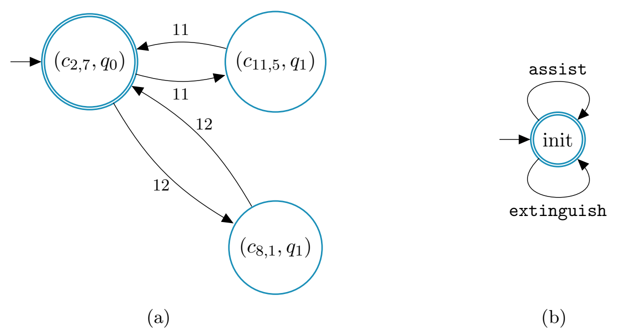


Fig. 3. (a) The global product automaton **G** that captures the motion of the quadrotor between the cells with static requests for the environment given in Figure 1 and the global mission specification ϕ_g (2). In a tuple corresponding to a state of **G**, the first element is a cell in the environment and the second element is a state of the Büchi automaton **B** that corresponds to ϕ_g. Here, $(c_{2,7}, q_0)$ is both the initial state and an accepting state of **G**. (b) The FSA **L** that corresponds to the regular expression ϕ_l in the local mission specification (3). Here **L** has a single state *init*, which is both the initial state and an accepting state. The labels on the edges show the dynamic requests that must be satisfied to take the corresponding transitions.

abort if there are no accepting states left in $\mathcal{F}_G$, meaning that the global mission specification ϕ_g cannot be satisfied. In line 4, we define a potential-like function $fd(q)$ that returns the shortest distance to the set of accepting states $\mathcal{F}_G$ for a state $q \in \mathcal{Q}_G$. The *shortest_dist*(q, q') function that we use here returns the length of the shortest path between states $q, q' \in \mathcal{Q}_G$ using Dijkstra's shortest path algorithm (Korte and Vygen, 2007). Note that the value returned by $fd(q)$ decreases as we get closer to the set of accepting states $\mathcal{F}_G$ and it returns zero once we are at an accepting state $q \in \mathcal{F}_G$. For a state $q \in \mathcal{Q}_G$ that cannot reach any accepting state in $\mathcal{F}_G$, $fd(q) = \infty$. As the global product automaton **G** may have multiple initial states due to the nondeterminism of **B**, in line 5 we set our current state in **G**, denoted by g_{cur}, to the state in $\mathcal{Q}_{G,init}$ that is closest to the set of accepting states $\mathcal{F}_G$. The final step of the *offline* part of our algorithm (line 6) is to construct the FSA **L** corresponding to the regular expression ϕ_l in the local mission specification and set our current state in **L**, denoted by l_{cur}, to the initial state $q_{L,init}$ of **L**.

Example 3.1 Revisited. *Figure 3(a) illustrates the global product automaton* **G** *that captures the motion of the quadrotor between the cells with static requests for the environment given in Figure 1 and the global mission specification* ϕ_g (2). *Figure 3(b) illustrates the FSA* **L** *that corresponds to the regular expression* ϕ_l *in the local mission specification* (3).

4.2. Online computation

The rest of Algorithm 1 (lines 8–33) is the *online* receding horizon control part that is executed every time the vehicle arrives at a cell and performs sensing, i.e. at each time step. In this part, our controller plans a local path within the sensing range of the vehicle that satisfies the local mission specification while taking the vehicle as close as possible towards satisfaction of the global mission specification.

Algorithm 2: Construct the local transition system **U**.

```
Input: x, y coordinates of the current cell of the
       vehicle.
Output: Local transition system U.
1 $\mathcal{Q}_U = \{c_{i,j} \mid |x-i| \le \frac{o-1}{2} \wedge |y-j| \le \frac{p-1}{2} \wedge c_{i,j} \in \mathcal{C}\}$.
2 $q_{U,init} = c_{x,y}$.
3 $\delta_U = \{(c_{i,j}, c_{k,l}) \mid c_{i,j}, c_{k,l} \in \mathcal{Q}_U \wedge ((|k-i| = 1 \wedge l = j) \vee (|l-j| = 1 \wedge i = k) \vee (i = k \wedge j = l))\}$.
4 $w_U(c_{i,j}, c_{k,l}) = 1 \ \forall (c_{i,j}, c_{k,l}) \in \delta_U$.
5 $h_U(c_{i,j}) = \mathcal{L}_d(t)(c_{i,j}) \ \forall c_{i,j} \in \mathcal{Q}_U$.
6 $\Pi_U = \{h_U(c_{i,j}) \mid c_{i,j} \in \mathcal{Q}_U\}$.
7 return U := $(\mathcal{Q}_U, q_{U,init}, \delta_U, \Pi_U, h_U, w_U)$.
```

First, we use Algorithm 2 to construct the local transition system **U** that models the motion of the vehicle between the cells within its sensing range and captures the dynamic requests sensed at each cell (Algorithm 1, line 8). We define the states of **U** as an $o \times p$ grid of square cells (the sensing grid discussed in Section 3.1) centered at the current cell $c_{x,y}$ of the vehicle and set its initial state to the current cell of the vehicle (Algorithm 2, lines 1–2). Next, we define unit weight transitions between all adjacent cells in $\mathcal{Q}_U$ and unit-weight self-loops for all cells in $\mathcal{Q}_U$ (Algorithm 2, lines 3–4). Then, we assign locally detected dynamic requests to their corresponding cells (Algorithm 2, lines 5–6).

We proceed by defining the set $\mathcal{D}_{service}$ of locally detected dynamic requests that we can service at the current state l_{cur} of **L** and the sets $\mathcal{Q}_s$ and $\mathcal{Q}_d$ of cells in the sensing range of the vehicle with static and dynamic requests, respectively (Algorithm 1, lines 10–11). If there are no serviceable dynamic requests within the sensing range of the vehicle, i.e. if $\mathcal{D}_{service} = \emptyset$, then the vehicle is driven towards the satisfaction of the global mission specification (Algorithm 1, lines 13–23). Otherwise, i.e. if $\mathcal{D}_{service} \neq \emptyset$, then the locally detected dynamic requests are satisfied (or avoided) according to the local mission specification (Algorithm 1, lines 25–29). In the first case, we consider all neighbors of our current state g_{cur} in **G**, denoted by g_{next} in line 13 of Algorithm 1, and define the corresponding sets of cells in $\mathcal{Q}_U$ that we want to reach and avoid, denoted by $\mathcal{Q}_{target}$ and $\mathcal{Q}_{avoid}$, respectively. If the cell corresponding to g_{next}, denoted by $g_{next}[0]$, is in the sensing range (Algorithm 1, lines 14–16), we set $\mathcal{Q}_{target}$ to this cell and define $\mathcal{Q}_{avoid}$ such that it consists of the cells with static and dynamic requests except this cell, i.e. $\mathcal{Q}_{avoid} = (\mathcal{Q}_s \cup \mathcal{Q}_d) \setminus \{g_{next}[0]\}$. If the cell corresponding to g_{next} is out of the sensing range (Algorithm 1, lines 17–19), we set $\mathcal{Q}_{avoid}$ to all of those cells with static and dynamic requests and drive the vehicle towards the boundary of the sensing range by setting $\mathcal{Q}_{target}$ to the boundary cells of **U** except those in $\mathcal{Q}_{avoid}$. Setting $\mathcal{Q}_{avoid}$ this way guarantees that the vehicle will not satisfy any unintended static or dynamic requests that can potentially violate the global or local mission specification as it goes towards $g_{next}[0]$, i.e. the cell corresponding to g_{next}. Then, in lines 20–23, we find the local trajectory that takes the vehicle closest to satisfaction of ϕ_g. Here, *shortest_path*$(q_{\mathrm{U},init}, \mathcal{Q}_{avoid})$ (Algorithm 1, line 20) returns the shortest paths from $q_{\mathrm{U},init}$ to all other cells in $\mathcal{Q}_\mathrm{U}$ that do not visit the cells in $\mathcal{Q}_{avoid}$ by temporarily setting the weights of the incoming edges of those cells to ∞ and *man_dist*(.) (Algorithm 1, line 22) returns the Manhattan distance between two given cells. In line 22, we calculate the predicted distance of the vehicle to the accepting states of **G** as the sum of $len(paths[q])$, $man\_dist(q, g_{next}[0])$, and $fd(g_{next})$. Line 23 keeps track of the next cell, denoted by $cell^\star_{next}$, of the best local plan obtained so far corresponding to the pair of $q \in \mathcal{Q}_{target}$ and $g_{next} \in \mathcal{Q}_\mathrm{G}$ that takes the vehicle closest to the set of accepting states of **G**. In our implementation, if there are multiple target cells from $\mathcal{Q}_{target}$ that result in the same d_{plan}, we use that with the smallest x coordinate and the largest y coordinate.

Lines 25–29 of Algorithm 1 correspond to the second case where there are serviceable dynamic requests within the sensing range of the vehicle, i.e. $\mathcal{D}_{service} \neq \emptyset$. Note that due to the priority function $prio : \mathcal{D} \to \mathbb{N}$, the vehicle must service only those dynamic requests from $\mathcal{D}_{service}$ with the highest priority. To this end, we first define our set of target cells $\mathcal{Q}_{target}$ as those cells having the highest priority dynamic requests from $\mathcal{D}_{service}$. Next, we define the set $\mathcal{Q}_{avoid}$ of cells that the vehicle must avoid as those cells with static or dynamic requests other than those in $\mathcal{Q}_{target}$. Then, in lines 27–29, we find the cell from $\mathcal{Q}_{target}$ that is closest to the current location of the vehicle and the next cell of the corresponding shortest path, denoted by $cell^\star_{next}$.

At the end of the planning stage discussed above, we abort if we cannot find a feasible local plan that both satisfies the local mission specification and drives the vehicle towards some neighbor g_{next} of g_{cur} that can reach $\mathcal{F}_\mathrm{G}$ (Algorithm 1, line 30). Otherwise, in lines 31–33, we update g_{cur} and l_{cur} as necessary and drive the vehicle to $cell^\star_{next}$. Once the vehicle reaches $cell^\star_{next}$, Algorithm 1 continues execution from line 8. Next, we state and prove the correctness of Algorithm 1 and discuss its *offline* and *online* complexities.

Theorem 4.2. *Assume that during its execution, Algorithm 1 reaches line 31 with* $cell^\star_{next} = g^\star_{next}[0]$ *and* $fd(g^\star_{next}) = 0$ *infinitely often. Then, the trajectory generated by the vehicle satisfies the global and local mission specifications.*

Proof. Algorithm 1 guides the vehicle between the cells with static requests by computing local plans. Under the given assumption, the controller satisfies the Büchi acceptance condition (Section 2) by visiting the set of accepting states of **G** infinitely often. Since we are also guaranteed not to visit any cells with static requests besides $g^\star_{next}[0]$, i.e. the cell corresponding to $g^\star_{next}$, due to the usage of the set $\mathcal{Q}_{avoid}$ during planning, the motion of the vehicle in the environment satisfies the global mission specification ϕ_g. Note also that any local plan computed using Algorithm 1 is guaranteed to satisfy the local mission specification because the vehicle always services the highest-priority dynamic request among those that are allowed at the current state of **L** corresponding to ϕ_l. Thus, under the given assumption, Algorithm 1 correctly solves Problem 3.2. ■

Remark 4.3 (Completeness). *The assumption employed in Theorem 4.2 corresponds to the cases where there is always at least one feasible local plan that both satisfies the local mission specification and gets the vehicle closer to the cell with the next static request that must be serviced to satisfy* ϕ_g. *One case where this assumption is trivially satisfied is when there are no dynamic requests and the cells with static requests are at least one cell apart. In this case, if the global mission* ϕ_g *is satisfiable, then a vehicle controlled by Algorithm 1 satisfies it. We may also consider the case where the rate at which dynamic requests appear is limited such that once a dynamic request is serviced, it cannot appear again until* g_{cur} *in Algorithm 1 reaches an accepting state of* **G**. *Then, if the cells with requests are at least one cell apart, Algorithm 1 solves Problem 3.2. However, mostly due to the dynamic nature of the locally detected requests and our assumption of limited sensing range, our approach is sound but not complete. There may be cases where the dynamic requests behave in an adversarial fashion resulting in the violation of the global and local mission specifications. For instance, a dynamic request that must be avoided may suddenly appear at the next cell of the vehicle or the locations of the dynamic requests may render the local specification unsatisfiable due to the limited sensing range of the vehicle. There may also be cases where a dynamic request may block the progress of the vehicle towards satisfaction of* ϕ_g *by reappearing continuously, essentially resulting in a livelock with respect to the global mission specification.*

Remark 4.4 (Offline and online complexity). *The Büchi automaton* **B** *corresponding to the global mission specification* ϕ_g *can be constructed in time* $O(2^{|\phi_g|})$, *where* $|\phi_g|$ *denotes the size of the formula measured in terms of the number of temporal and Boolean operators in* ϕ_g *(Baier and Katoen, 2008). The worst-case size of* **B** *is also* $O(2^{|\phi_g|})$ *(Baier and Katoen, 2008), making the worst-case size of the global product automaton* **G** $O(|\mathcal{Q}_\mathrm{T}|2^{|\phi_g|})$, *where* $|\mathcal{Q}_\mathrm{T}|$ *denotes the number of states in* **T**. *Thus, constructing the function* $fd(.)$ *takes time polynomial in* $|\mathcal{Q}_\mathrm{T}|$ *and exponential in* $|\phi_g|$. *Since converting the regular expression* ϕ_l *to a deterministic FSA can be done in time* $O(2^{|\phi_l|})$, offline *complexity of our algorithm is polynomial in* $|\mathcal{Q}_\mathrm{T}|$ *and exponential in* $|\phi_g|$ *and* $|\phi_l|$. *The* online *complexity of our algorithm depends on whether there are dynamic requests within the sensing range of the vehicle or not. If there are no dynamic requests, lines 13–23 of Algorithm 1 take time exponential in* $|\phi_g|$ *and polynomial in* $|\mathcal{Q}_\mathrm{T}|$ *and* $|\mathcal{Q}_\mathrm{U}|$. *If there are dynamic requests in the sensing range, lines 25–29 of Algorithm 1 take time polynomial in* $|\mathcal{Q}_\mathrm{U}|$. *In typical cases where* *the size of the local transition system* **U** *is larger than the global product automaton, the* online *complexity of our algorithm is polynomial in* $|\mathcal{Q}_{\mathrm{U}}|$. *We must also note that the size of the transition system* **T** *used by our algorithm scales only with the number of cells with static requests and not with the overall size of the environment as in Kress-Gazit et al. (2007), Wongpiromsarn et al. (2010), Ding et al. (2012), Maly et al. (2013), and Kim and Fainekos (2013).*

5. Implementation and case studies

In this section we discuss the details of our implementation and experimental setup after which we present the results of our experiments and simulations.

5.1. Software implementation and experimental setup

The controller presented in Algorithm 1 is implemented as a Python module that returns the next cell the quadrotor must fly to given its current location in the environment.[1] In our implementation, we use ltl2ba (Gastin and Oddoux, 2001) to obtain the Büchi automaton **B** corresponding to ϕ_g, and use the dk.brics.automaton package (Møller, 2011) to obtain the FSA **L** corresponding to ϕ_l (Algorithm 1). Our implementation can be used either to simulate the trajectory of the vehicle in a virtual environment or to control the vehicle in an experimental setup. In the latter case, our implementation uses the measurements provided by the experimental platform to compute the next cell the vehicle must go to. The low-level controllers that control the vehicle from one grid cell to another according to the output of Algorithm 1 are implemented in Matlab. We discuss these low-level quadrotor controllers in Section 5.2.

Our experimental platform comprises four Viewsonic short-throw projectors, an Optitrack motion capture system, a kQuad500 quadrotor from KMel Robotics equipped with a camera facing downwards, and three desktop computers communicating over a local area network. The motion capture system tracks the motion of the quadrotor and provides the low-level controllers with accurate position information. The projectors project the color-coded cells representing the static and dynamic requests on the ground, which are sensed by the quadrotor within its local sensing range. The computers run the code responsible for trajectory planning (Algorithm 1) and low-level control of the quadrotor as well as processing the images sent by the quadrotor.

5.2. Quadrotor low-level controller

To navigate in the environment, the quadrotor must perform a two-step sequence in which it first flies from its current cell to one of its neighboring cells as given by Algorithm 1, then remains in the target cell while performing the necessary sensing and planning operations. The low-level quadrotor controller that ensures a safe and smooth transition from one cell to another is based on:

(i) input–output linearization for the quadrotor dynamics (Voos, 2009); and
(ii) construction of multi-affine vector fields for control-to-facet and invariance in rectangles (Belta and Habets, 2006; Kloetzer and Belta, 2006).

Trajectory planning between adjacent cells is carried out by creating vector fields within the current and target cells of the quadrotor, which map locations in the cells to desired velocity vectors. In the current cell, the vector field is computed such that it guarantees all trajectories starting in the initial cell pass through the connecting facet and into the target cell. Within the target cell, the vector field is computed to guarantee the convergence of all trajectories to the center of the cell preventing the quadrotor from leaving through any of its surrounding facets once it arrives in the cell (Belta and Habets, 2006; Kloetzer and Belta, 2006). Figure 4 shows a close-up of the vector fields of two adjacent cells. Note that the vector field is continuous everywhere in the region spanned by the two rectangles, which implies that the corresponding trajectory is smooth.

The vector field construction described above provides a function $v : \mathbb{R}^3 \to \mathbb{R}^3$ that maps the measured position to a desired velocity. Feedback control around a velocity setpoint is achieved by separating the control of the translational and rotational motions of the quadrotor into an outer and inner control loop. The outer loop first calculates the desired acceleration $\ddot{\mathbf{x}}^d$ based on proportional–integral feedback with respect to the velocity setpoint, and then transforms the control outputs into a desired attitude. A feedback linearization approach is used to find the desired attitude by solving the dynamic equations for an orientation and thrust that produces the appropriate motion, while accounting for the non-linear dynamics in the model (Voos, 2009). More specifically, we define a coordinate frame F_G fixed to the ground and a coordinate frame F_B fixed to the body of the quadrotor with its origin at the center of mass. The body frame is defined with respect to the ground frame by the position vector $\mathbf{x}$ and by Z–X–Y Euler angles, denoted by α. Using Euler–Lagrange equations, the translational acceleration of the body frame in the ground frame can be expressed as a function of the orientation and thrust of the quadrotor (Mellinger and Kumar, 2011):

$$m\,\ddot{\mathbf{x}} = -m\,g\,\mathbf{z}_G + f_T\,\mathbf{R}\,\mathbf{z}_B, \tag{4}$$

where m is the mass of the vehicle, g is the gravitational acceleration, $\mathbf{z}_G$ is the z unit vector in F_G, f_T is the total thrust produced by the four rotors, $\mathbf{R}$ is the rotation matrix to express vectors in the body frame with respect to the ground frame, and $\mathbf{z}_B$ is the z unit vector in F_B. The rotational motion of the quadrotor can be expressed as a function of the net moments, $\mathbf{M}_\alpha$, about each of the body frame axes using Euler's equation (Mellinger and Kumar, 2011):

$$\mathcal{I}\,\dot{\omega} = \mathbf{M}_\alpha - \omega \times \mathcal{I}\,\omega, \tag{5}$$

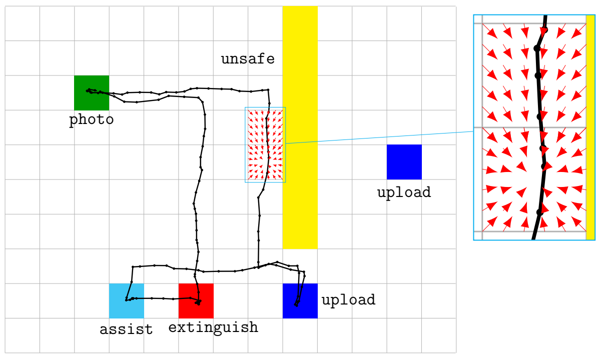


Fig. 4. Quadrotor trajectory plotted over the environment in Example 3.1. The close-up on cells $c_{7,6}$ and $c_{7,5}$ show the vector fields computed for flying the quadrotor from $c_{7,6}$ to $c_{7,5}$.

where ω is the angular velocity of the body frame with respect to the ground frame and $\mathcal{I}$ is the inertia matrix with respect to the body frame. Then, the desired acceleration $\ddot{\mathbf{x}}^d$ given by the vector field can be substituted in place of $\ddot{\mathbf{x}}$ in the dynamics equation given in (4) to create a system of equations which can be solved in closed form for an expression for an attitude and total thrust that will produce the desired acceleration. The desired attitude is then used as a reference for the inner control loop, designed to stabilize the attitude of the quadrotor. Attitude stabilization of the quadrotor is performed by the inner loop controller, where inertial measurement unit (IMU) measurement data is utilized to stabilize the quadrotor at the attitude setpoint resulting from the outer loop controller. A feedback linearization approach can be used to account for the non-linear dynamics of the system:

$$\mathbf{M}_\alpha = \mathcal{I}(\mathbf{k}_{\mathbf{p},\alpha}\mathbf{e}_\alpha + \mathbf{k}_{\mathbf{d},\alpha}\dot{\mathbf{e}}_\alpha) + \omega \times \mathcal{I}\omega, \qquad (6)$$

where e_α is the attitude error and $k_{p,\alpha}$ and $k_{d,\alpha}$ are gain parameters. The resulting architecture is a nested loop of controllers capable of stabilizing the quadrotor at any given velocity setpoint. Our extensive experimental testing showed that this approach produces very satisfactory results. The trajectory of the center of the quadrotor stays close to the middle of the traversed rectangles for all times (see Figure 4).

5.3. Case studies

5.3.1. Case study 1 We return to the persistent surveillance mission given in Example 3.1 and present the results of our experiments where a quadrotor satisfies the global mission specification (2) and the local mission specification (3). Figure 4 illustrates the trajectory followed by the quadrotor for a particular realization of dynamic requests during the experiments. In Figure 4, the black line corresponds to the actual flight path of the quadrotor over the environment as provided by the motion capture system. Figure 5 shows four snapshots from a video of the experimental trial shot with an overhead camera, which correspond to four instances of the trajectory given in Figure 4. The red lines in Figure 5 indicate the sequence of cells traversed by the quadrotor. Note that at the beginning of the flight, only the `unsafe` dynamic request is there (see Figure 5(a)). The remaining dynamic requests `extinguish` and `assist` appear later in the flight as we discuss in the following. The quadrotor begins its flight at $c_{2,7}$ servicing the static `photo` request. Next, the quadrotor has to service the static `upload` request at either $c_{8,1}$ or $c_{11,5}$. As the quadrotor cannot detect the unsafe cells yet and $c_{11,5}$ is closer to its current position than $c_{8,1}$, it starts flying towards $c_{11,5}$. Note that since the `unsafe` dynamic request does not appear in ϕ_l, the quadrotor must avoid it at all times. Once the quadrotor reaches $c_{7,7}$ it can no longer go east due to the unsafe cells and can only fly south to get closer to $c_{11,5}$. When the quadrotor arrives at $c_{7,5}$, the controller finds that flying towards $c_{8,1}$ takes the quadrotor closer to satisfying ϕ_g than flying to $c_{11,5}$ does, so the quadrotor starts flying towards $c_{8,1}$ to service the `upload` request. Figure 5(a) shows the sequence of cells traversed by the quadrotor between the static `photo` request at $c_{2,7}$ and the static `upload` request at $c_{8,1}$. After reaching $c_{8,1}$, the quadrotor needs to fly back to $c_{2,7}$ to service the `photo` request as required by ϕ_g. As the quadrotor leaves $c_{8,1}$ for $c_{2,7}$, the `assist` and `extinguish` dynamic requests appear at $c_{3,1}$ and $c_{5,1}$, respectively (see Figure 5(b)). However, due to its limited sensing range, the quadrotor only detects the `extinguish` request and starts flying west to reach $c_{5,1}$. At $c_{5,2}$, the quadrotor detects the `assist` request and flies to $c_{3,1}$ as assisting a survivor is of higher priority than extinguishing a fire according to the local mission specification. Figure 5(b) shows the sequence of cells traversed by the quadrotor between the static `upload` request at $c_{8,1}$ and the dynamic `assist` request at $c_{3,1}$. After assisting the survivor at $c_{3,1}$, the quadrotor extinguishes the fire at $c_{5,1}$ and flies to $c_{2,7}$ to service the `photo` request (see Figures 5(c) and 5(d)). Note that the `assist` and `extinguish` dynamic requests disappear once they are serviced. As the quadrotor performs a persistent surveillance mission, it keeps servicing `photo` and `upload` requests indefinitely while responding to locally detected dynamic requests according to the local mission specification. The trajectory shown in Figure 4 is a portion of the infinite mission of the quadrotor. Extension 1 shows the actual flight of the quadrotor in our experimental setup. During the experiments, maximum execution times of the offline and online portions of Algorithm 1 were 15 and 5 ms, respectively, when executed on an iMac i5 quad-core computer. Total flight time of the quadrotor for the trajectory shown in Figure 4 was approximately 90 seconds.

5.3.2. Case study 2 Next, we present the simulation results of another surveillance mission on a larger 23×14 environment illustrated in Figure 6. The quadrotor that we consider in this case study has a 7×7 sensing range and starts at

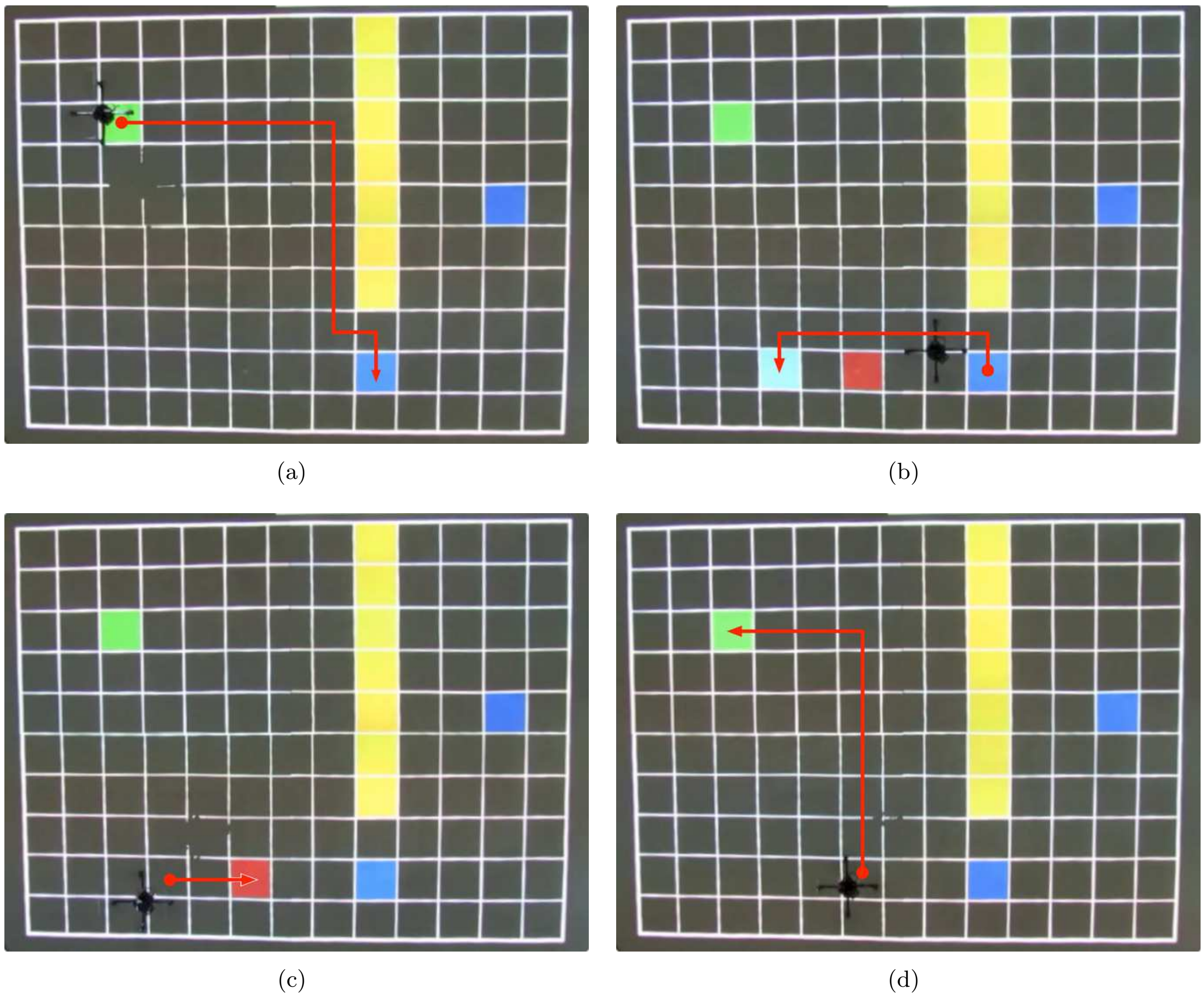


Fig. 5. Four snapshots from a video of the experimental trial shot with an overhead camera, which correspond to four instances of the trajectory given in Figure 4. The red lines indicate the sequence of cells traversed by the quadrotor.

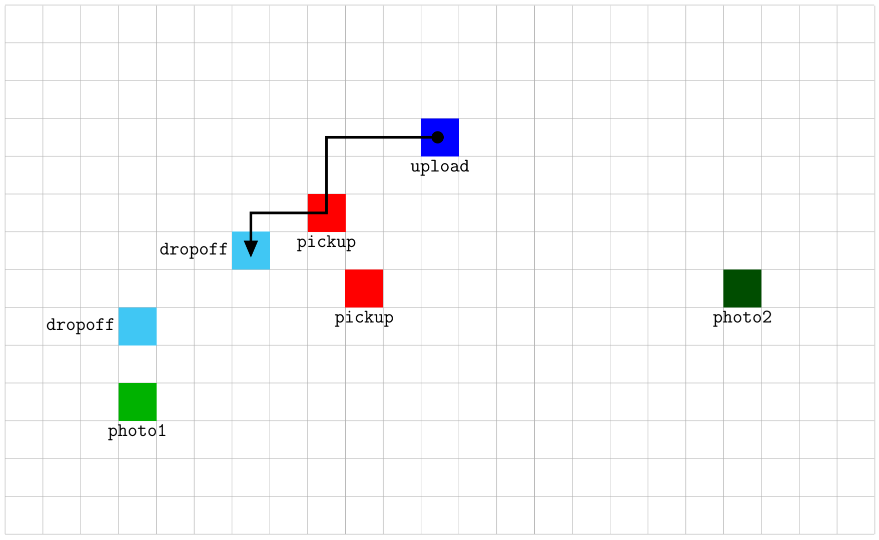
$c_{3,3}$. We have three static requests defined in the environment with $\mathcal{S} = \{\texttt{photo1}, \texttt{photo2}, \texttt{upload}\}$, $\mathcal{L}_s(c_{3,3}) = \texttt{photo1}$, $\mathcal{L}_s(c_{19,6}) = \texttt{photo2}$, and $\mathcal{L}_s(c_{11,10}) = \texttt{upload}$. Cells $c_{8,8}$, $c_{9,6}$ and $c_{6,7}$, $c_{3,5}$ are shown in red and cyan; and correspond to the locally detected `pickup` and `dropoff` dynamic requests, respectively. The global mission of the vehicle is to "repeatedly take photos at `photo1` followed by `photo2` and to `upload` current photos before taking new photos" which can be expressed in LTL as

$$\begin{aligned} \phi_g := {} & \mathbf{GF}\texttt{photo1} \wedge \mathbf{G}(\texttt{photo1} \Rightarrow \mathbf{X}\texttt{photo2}) \wedge \\ & \mathbf{G}(\texttt{photo2} \Rightarrow \mathbf{X}\texttt{upload}) \wedge \\ & \mathbf{G}(\texttt{upload} \Rightarrow \mathbf{X}\texttt{photo1}). \end{aligned} \tag{7}$$

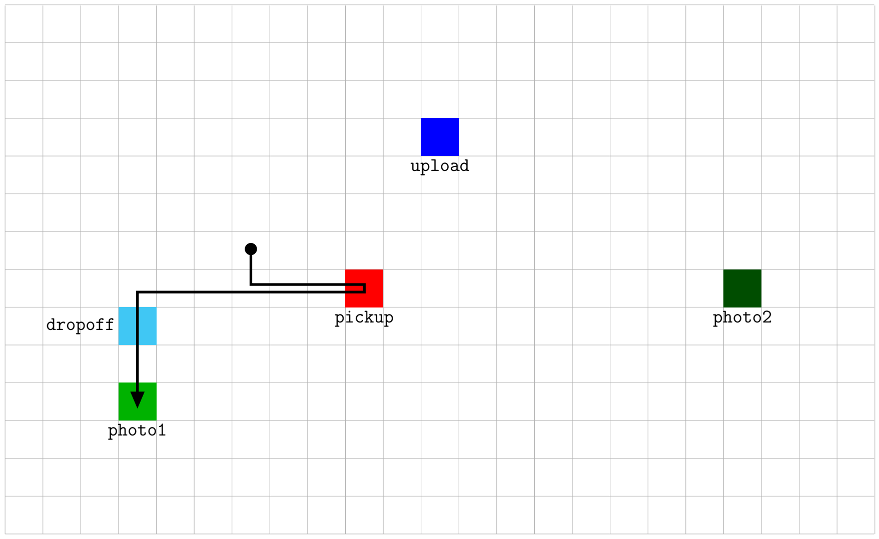
The local mission specification is to "`pickup` and `dropoff` locally detected items one at a time as they are discovered and to service `dropoff` only if carrying an item" which translates to the local mission specification

$$\begin{aligned} & \phi_l := (\texttt{pickup}.\texttt{dropoff})^* \\ & \mathit{prio}(\texttt{pickup}) = 0,\ \mathit{prio}(\texttt{dropoff}) = 0. \end{aligned} \tag{8}$$

In the following, we discuss the trajectory of the quadrotor between the static `upload` request at $c_{11,10}$ and the static `photo1` request at $c_{3,3}$ as illustrated in Figure 6. When the quadrotor arrives at $c_{9,10}$ after servicing the `upload` request, the `pickup` and `dropoff` dynamic requests appear. At this point, the quadrotor can sense only the `pickup` request at $c_{8,8}$ and the `dropoff` request at $c_{6,7}$. Since the quadrotor cannot dropoff before picking up due to ϕ_l, it first services the dynamic `pickup` request at $c_{8,8}$. At $c_{8,8}$, the quadrotor can sense both the `pickup` request at $c_{9,6}$ and the `dropoff` request at $c_{6,7}$, but can only service the `dropoff` request due to ϕ_l. Thus, the quadrotor proceeds to $c_{6,7}$ to drop its current cargo off. Figure 6(a) shows the sequence of cells traversed by the quadrotor between the static `upload` request at $c_{11,10}$ and the dynamic `dropoff` request at $c_{6,7}$. After dropping its cargo off at $c_{6,7}$, the quadrotor goes to $c_{9,6}$ to service the `pickup` request. When the quadrotor is at $c_{9,6}$, it can no longer sense the `dropoff` request at $c_{3,5}$ and starts going west to service the static `photo1` request as required by ϕ_g. As the quadrotor goes west, the `dropoff` request at $c_{3,5}$ is detected again



(a)

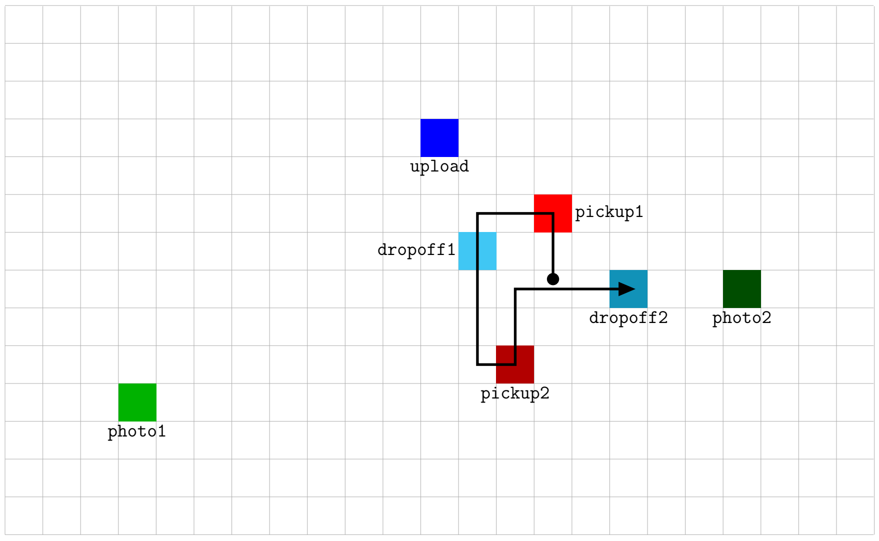


(b)

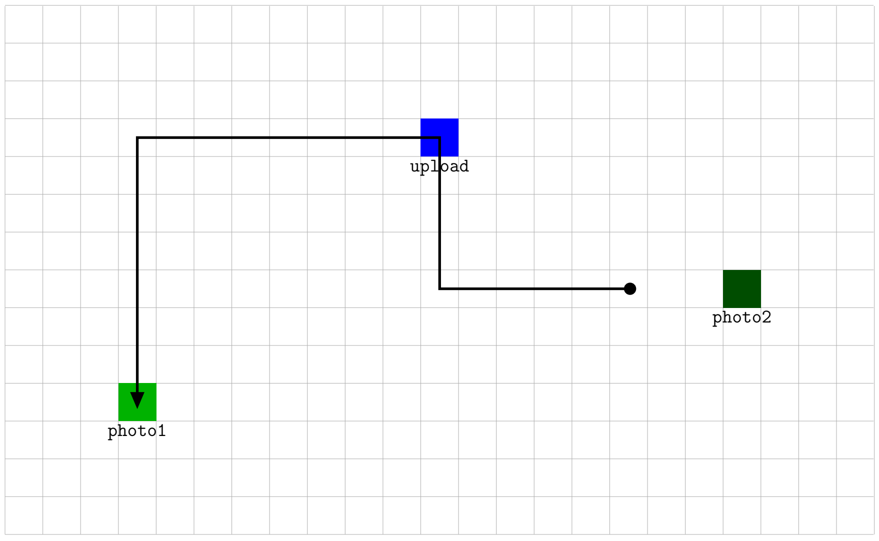
Fig. 6. Parts of the trajectory followed by the quadrotor in case study 2 as it flies from `upload` to `photo1`. We have three static requests defined in the environment with $\mathcal{S} = \{\texttt{photo1}, \texttt{photo2}, \texttt{upload}\}$, $\mathcal{L}_s(c_{3,3}) = \texttt{photo1}$, $\mathcal{L}_s(c_{19,6}) = \texttt{photo2}$, and $\mathcal{L}_s(c_{11,10}) = \texttt{upload}$. Cells $c_{8,8}$, $c_{9,6}$ and $c_{6,7}$, $c_{3,5}$ are shown in red and cyan; and correspond to the locally detected `pickup` and `dropoff` dynamic requests, respectively.

and the quadrotor services `dropoff` at $c_{3,5}$ before servicing `photo1` at $c_{3,3}$. Figure 6(b) shows the sequence of cells traversed by the quadrotor between the dynamic `dropoff` request at $c_{6,7}$ and the static `photo1` request at $c_{3,3}$. Extension 1 shows the trajectory followed by the quadrotor during a full cycle between `photo1`, `photo2`, and `upload`. During the simulations, maximum execution times of the offline and online portions of Algorithm 1 were 15 and 7 ms, respectively, when executed on an iMac i5 quad-core computer.

5.3.3. Case study 3 Now we consider the same environment and global mission specification (7) as in the previous case study with a richer local mission specification. More specifically, we have two types of cargo that can be picked up and dropped off by servicing dynamic requests `pickup1`, `dropoff1` and `pickup2`, `dropoff2`, respectively. Similar to the previous case study, the quadrotor can carry only one cargo at a time, but this time items of type 1 have higher priority than those of type 2. This local mission specification translates to



(a)



(b)

Fig. 7. Parts of the trajectory followed by the quadrotor in case study 3 as it flies from $c_{14,6}$ to photo1.

$$\phi_l := (\texttt{pickup1.dropoff1}|\texttt{pickup2.dropoff2})^*$$
$$prio(\texttt{pickup1}) = 0,\ prio(\texttt{dropoff1}) = 0,$$
$$prio(\texttt{pickup2}) = 1,\ prio(\texttt{dropoff2}) = 1. \quad (9)$$

In the following, we discuss the trajectory of the quadrotor between $c_{14,6}$ and the static photo1 request at $c_{3,3}$ as illustrated in Figure 7. When the quadrotor arrives at $c_{14,6}$, the dynamic requests pickup1, dropoff1, pickup2, and dropoff2 appear at $c_{14,8}$, $c_{12,7}$, $c_{13,4}$, and $c_{16,6}$, respectively. At this point, all four dynamic requests are within the sensing range of the quadrotor (a 7×7 grid). However, due to ϕ_l, only the pickup requests can be serviced as the quadrotor does not have any cargo to drop off. Also, the local mission specification gives higher priority to pickup1. Thus, the quadrotor goes to $c_{14,8}$ to pick cargo of type 1 up after which it goes to $c_{12,7}$ to drop if off. Since after servicing dropoff1 the quadrotor has no cargo, it goes to $c_{13,4}$ to pick the cargo of type 2 up and drops it off at $c_{16,6}$. Figure 7(a) shows the sequence of cells traversed by the quadrotor between $c_{14,6}$ and the dynamic dropoff2 request at $c_{16,6}$. After this, no dynamic requests remain, and the quadrotor proceeds to $c_{11,10}$ to service the static upload request followed by the static photo1 request at $c_{3,3}$. Figure 7(b) shows the sequence of cells traversed by the quadrotor between the dynamic dropoff2 request at $c_{16,6}$ and the static photo1 request at $c_{3,3}$. Extension 1 shows the trajectory followed by the quadrotor during a full cycle between photo1, photo2, and upload. During the simulations, maximum execution times of the offline and online portions of Algorithm 1 were 15 and 7 ms, respectively, when executed on an iMac i5 quad-core computer.

6. Conclusion

We have presented a computational framework for automatic synthesis of a control strategy driving an autonomous vehicle through the regions of a partitioned environment while satisfying rich, temporal logic specifications over service requests. The main contribution of the paper is to show that temporal logic specifications over static requests with known locations can be satisfied while dynamic requests sensed locally are serviced as well. Our receding horizon implementation of controllers ensures fast responses to rapidly changing local requests. We demonstrated the applicability of our approach with experiments and simulations involving a quadrotor performing a persistent surveillance task over a planar grid environment.

The main advantage of our framework is its computational efficiency. Since the planning is done locally in a reactive fashion, the complexity of our approach scales with the size of the local sensing range and the number of static requests as opposed to the overall size of the environment, which can be potentially very large. However, this advantage comes at a cost: dynamic requests may behave in an adverserial fashion blocking the progress of the vehicle or some low-priority dynamic request may go unserviced if it gets out of the sensing range as the vehicle moves towards a higher-priority dynamic request. It seems like one may alleviate these issues by adding a layer that *remembers* the dynamic requests detected by the vehicle. However, this should be done carefully as it may substantially increase the complexity of the approach. If the vehicle keeps track of all detected dynamic requests, then the planning may eventually have to be performed in a graph as large as the overall environment.

Funding

This work was supported in part by the Office of Naval Research (grant numbers MURI N00014-09-1051 and MURI N00014-10-10952) and by the National Science Foundation (grant number CNS-1035588).

Note

1. Our implementation is available online at http://hyness.bu.edu/rhtlc/.

References

Baier C and Katoen JP (2008) *Principles of Model Checking*. Cambridge, MA: MIT Press.

Belta C and Habets LCGJM (2006) Control of a class of nonlinear systems on rectangles. *IEEE Transactions on Automatic Control* 51(11): 1749–1759.

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

Bhatia A, Kavraki LE and Vardi M (2010) Sampling-based motion planning with temporal goals. In: *IEEE international conference robotics and automation*, Anchorage, AK, USA, pp. 2689–2696.

Ding XC, Kloetzer M, Chen Y and Belta C (2011) Formal methods for automatic deployment of robotic teams. *IEEE Robotics and Automation Magazine* 18: 75–86.

Ding XC, Lazar M and Belta C (2012) Receding horizon temporal logic control for finite deterministic systems. In: *American control conference*, Montreal, Canada.

Gastin P and Oddoux D (2001) Fast LTL to Büchi automata translation. In: *Proceedings of the 13th international conference on computer aided verification (CAV'01)*, Paris, France (*Lecture Notes in Computer Science*, vol. 2102). New York: Springer, pp. 53–65.

Karaman S and Frazzoli E (2011) Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research* 30(7): 846–894.

Kim K and Fainekos G (2013) Minimal specification revision for weighted transition systems. In: *IEEE international conference robotics and automation*, pp. 4068–4074.

Kloetzer M and Belta C (2006) A framework for automatic deployment of robots in 2D and 3D environments. In: *IEEE/RSJ international conference intelligent robots and systems*, pp. 953–958.

Korte B and Vygen J (2007) *Combinatorial Optimization: Theory and Algorithms* (*Algorithmics and Combinatorics*, vol. 21), 4th edn. New York: Springer.

Kress-Gazit H, Fainekos G and Pappas GJ (2007) Where's Waldo? Sensor-based temporal logic motion planning. In: *IEEE international conference robotics and automation*, pp. 3116–3121.

Kress-Gazit H, Wongpiromsarn T and Topcu U (2011) Correct, reactive robot control from abstraction and temporal logic specifications. *IEEE Robotics and Automation Magazine* 18: 65–74.

Maly MR, Lahijanian M, Kavraki LE, Kress-Gazit H and Vardi MY (2013) Iterative temporal motion planning for hybrid systems in partially unknown environments. In: *ACM international conference on hybrid systems: computation and control*, pp. 353–362.

Mellinger D and Kumar V (2011) Minimum snap trajectory generation and control for quadrotors. In: *IEEE international conference robotics and automation*, Shangai, China.

Møller A (2011) dk.brics.automaton – Finite-state automata and regular expressions for Java. http://www.brics.dk/automaton/.

Rodger SH and Finley TW (2006) *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers.

Ulusoy A, Marrazzo M and Belta C (2013a) Receding horizon control in dynamic environments from temporal logic specifications. In: *Robotics: Science and Systems*, Berlin, Germany.

Ulusoy A, Marrazzo M, Oikonomopoulos K, Hunter R and Belta C (2013b) Temporal logic control for an autonomous quadrotor in a nondeterministic environment. In: *IEEE international conference robotics and automation*, Karlsruhe, Germany.

Voos H (2009) Nonlinear control of a quadrotor micro-UAV using feedback-linearization. In: *IEEE international conference mechatronics*, pp. 1–6.

Wongpiromsarn T, Topcu U and Murray RM (2010) Receding horizon control for temporal logic specifications. In: *Hybrid systems: computation and control*, Stockholm, Sweden, pp. 101–110.

Appendix: Index to Multimedia Extension

The multimedia extension page is found at http://www.ijrr.org

Table of Multimedia Extension

Extension	Media type	Description
1	Video	Quadrotor trajectories of case studies 1, 2, and 3.