Receding Horizon Control in Dynamic Environments from Temporal Logic Specifications

Alphan Ulusoy, Michael Marrazzo, and Calin Belta
Division of Systems Engineering, Boston University, Brookline, MA, 02446
{alphan, marrazzo, cbelta}@bu.edu

***Abstract*—We present a control strategy for an autonomous vehicle that is required to 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, and a servicing priority order over a set of dynamic requests that can be sensed locally. 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 in an experimental setup consisting of a quadrotor performing a persistent surveillance task over a planar grid environment.**

I. Introduction

Temporal logics have been traditionally used to specify the correctness of computer programs [1]. They recently gained popularity in robotics due to their ability to express complex robotics tasks [17, 11, 4, 8, 5, 12]. 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 [1] can be used to automatically generate motion plans and control strategies.

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 problem when dynamical events sensed locally are part of the mission specification [11, 17]. Consider, for example, a persistent surveillance mission in a disaster area, in which an autonomous flying vehicle is required to keep on photographing known damage areas. At the same time, the vehicle is required to look for survivors, fires, and gas leakages, which can be sensed locally around the vehicle using onboard sensors. The goal is to be able to react correctly to the events sensed locally (e.g., extinguish fires, provide medical assistance to survivors), while at the same time surveying the known damage areas.

Inspired by this scenario, in this paper 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, and a *local* specification given as a servicing priority order over *dynamic* requests sensed locally. 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 the 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 [2]. The main contribution of this work is the control synthesis algorithm that satisfies both the global (static) and local (dynamic) temporal logic specifications. Another contribution is the successful implementation of this computational framework in an experimental setup involving an autonomous quadrotor flying in an indoor environment.

This work is closest related to [11, 17, 6]. In [11, 17], the authors propose to deal with dynamic environments by using a fragment of Linear Temporal Logic (LTL) called General Reactivity (1) (GR(1)) as a specification language. Even though the control policies obtained using these methods are reactive, the synthesis algorithms have to take all dynamic events into account, resulting in a massive state space significantly hindering their scalability. In contrast, our approach initially considers only static requests and plans for dynamic requests only within the local sensing range and only as required. In [6], the authors 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. As opposed to this paper, in which complexity scales only with the static requests and the local sensing range of the robot, the algorithms from [6] scale with the size of the overall environment. Furthermore, our approach considers richer local specifications that both allow ordering among dynamic requests as well as avoiding them, whereas the approach given in [6] considers only maximization of collected rewards and cannot be trivially modified to obtain such rich local behaviors.

The rest of the paper is organized as follows. In Sec. II, we give necessary definitions and preliminaries in formal methods. In Sec. III, we formally state the problem that we consider in this paper, and present our solution in Sec. IV. We

present our experimental results in V. We conclude with final remarks in Sec. VI.

II. 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 [1] 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 II.1 (Transition System). *A (weighted, deterministic) transition system is a tuple* $\mathbf{T} := (\mathcal{Q}, q_{init}, \delta, \Pi, h, w)$*, where*

- $\mathcal{Q}$ *is the finite set of states;*
- $q_{init} \in \mathcal{Q}$ *is the initial state;*
- $\delta \subseteq \mathcal{Q} \times \mathcal{Q}$ *is the transition relation;*
- Π *is the finite set of atomic propositions;*
- $h : \mathcal{Q} \to 2^{\Pi}$ *is the labeling function giving the set of atomic propositions satisfied at a state;*
- $w : \delta \to \mathbb{N}$ *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 $q^0 = q_{init}$, $q^k \in \mathcal{Q}$, and $(q^k, q^{k+1}) \in \delta$ for all k. A run generates a *word* $\omega^0\omega^1, \ldots$, where $\omega^k = h(q^k)$ is the set of atomic propositions satisfied at state q^k.

Linear Temporal Logic (LTL) is an extension of (Boolean) propositional logic that can capture temporal relations [1]. LTL formulas are interpreted over infinite words generated by the transition system $\mathbf{T}$ from Def. II.1. Informally, an LTL formula ϕ over a set of atomic propositions Π can involve Boolean operators $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), and temporal operators $\Box$ (always), $\Diamond$ (eventually), $\bigcirc$ (next), and $\mathcal{U}$ (until). For instance, $\Box \mathtt{p}$ states that $\mathtt{p}$ is true at all positions of the word, $\Diamond \mathtt{p}$ states that $\mathtt{p}$ eventually becomes true in the word, and $\bigcirc \mathtt{p}$ states that proposition $\mathtt{p}$ is true at the next position of the word. Formula $\mathtt{p}_1 \mathcal{U} \mathtt{p}_2$ states that there exists $k \geq 0$ such that w^k satisfies $\mathtt{p}_2$ and w^j satisfies $\mathtt{p}_1$ for all $0 \leq j < k$, where w^k is the symbol at the k^{th} 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 II.2 (Büchi Automaton). *A Büchi automaton is a tuple* $\mathbf{B} := (\mathcal{Q}, \mathcal{Q}_{init}, \delta, \Sigma, \mathcal{F})$*, where*

- $\mathcal{Q}$ *is the finite set of states;*
- $\mathcal{Q}_{init} \subseteq \mathcal{Q}$ *is the set of initial states;*
- $\delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ *is the (non-deterministic) transition relation;*
- Σ *is the input alphabet;*
- $\mathcal{F} \subseteq \mathcal{Q}$ *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 $q^0 \in \mathcal{Q}_{init}$, and $(q^k, \omega^k, q^{k+1}) \in \delta$, for all k. A Büchi automaton $\mathbf{B}$ accepts a word over Σ if and only if at least one of the corresponding runs intersects with $\mathcal{F}$ infinitely many times. For any LTL formula ϕ over a set Π, one can construct a Büchi automaton with input alphabet $\Sigma = 2^{\Pi}$ accepting all and only words over 2^{Π} that satisfy ϕ using automated tools such as *ltl2ba* given in [7].

In this paper, we consider specifications expressed in two particular subclasses of LTL, namely, $\text{LTL}_{-\mathbf{X}}$ [1] and syntactically co-safe LTL (scLTL) [13]. $\text{LTL}_{-\mathbf{X}}$ is LTL without the $\bigcirc$ (next) operator and is typically used in settings where one is interested in stutter-invariant properties, i.e. the exact number of immediate repetitions of a symbol is irrelevant [15, 1]. A syntactically co-safe LTL (scLTL) formula excludes the $\Box$ (always) operator and the $\neg$ (negation) operator appears only in front of the atomic propositions when written in positive normal form [13]. A special property of scLTL is that one can determine if a given infinite word satisfies an scLTL formula by considering only a finite prefix of it [13].

Definition II.3 (Finite State Automaton). *A (deterministic) finite state automaton (FSA) is a tuple* $\mathbf{A} := (\mathcal{Q}, q_{init}, \delta, \Sigma, \mathcal{F})$*, where*

- $\mathcal{Q}$ *is the finite set of states;*
- $q_{init} \in \mathcal{Q}$ *is the initial state;*
- Σ *is the input alphabet;*
- $\delta : \mathcal{Q} \times \Sigma \times \mathcal{Q}$ *is the deterministic transition relation;*
- $\mathcal{F} \subseteq \mathcal{Q}$ *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 $q^0 = q_{init}$ and $(q^k, \omega^k, q^{k+1}) \in \delta$ for all k. An FSA $\mathbf{A}$ accepts a word of length n over Σ if and only if the corresponding run ends in some $q^n \in \mathcal{F}$. For any scLTL formula ϕ over Π, one can construct an FSA with input alphabet $\Sigma = 2^{\Pi}$ accepting all and only the finite words over 2^{Π} that satisfy ϕ using automated tools such as *scheck* given in [14].

III. Problem Formulation and Approach

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 (Rem. III.3), we show that this is enough for a large class of problems including vehicles with non-trivial dynamics. In Sec. V-B, we present the details of such an implementation for an autonomous quadrotor.

Formally, the problem is formulated as follows. Let

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

be a (planar) grid environment, where $\mathcal{C} = \{c_{x,y} | 0 \leq x < m, 0 \leq y < n\}$ 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, $\mathcal{L}_s : \mathcal{C} \to \mathcal{S}$ 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 $\mathcal{L}_d(t) : \mathcal{C} \to \mathcal{D}$ is a time varying (possibly partial) map that gives the location of the dynamic requests on the grid. We assume that the cells in the environment are arranged such 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 (Fig. 1).

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; and 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 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.

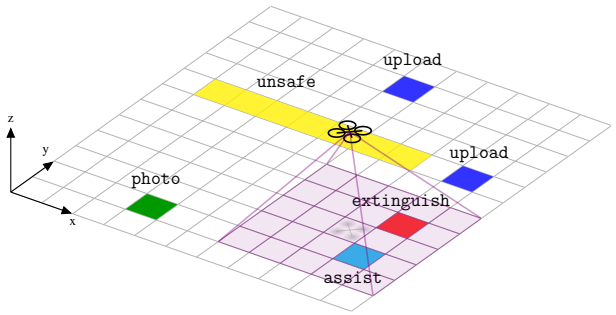

Fig. 1. A schematic representation of the scenario considered in Exp. III.1. We have two static requests defined in the environment with $\mathcal{S} = \{\texttt{photo}, \texttt{upload}\}$, $\mathcal{L}_s(c_{3,1}) = \texttt{photo}$, $\mathcal{L}_s(c_{5,10}) = \texttt{upload}$, and $\mathcal{L}_s(c_{9,7}) = \texttt{upload}$. The cells within the sensing range of the quadrotor are highlighted in violet. Cells $c_{9,4}$ and $c_{9,2}$ are shown in red and cyan; and correspond to locally detected `extinguish` and `assist` dynamic requests, respectively.

Example III.1. *Fig. 1 gives a schematic representation of an* 11×12 *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* $\mathcal{L}_s(c_{3,1}) = \texttt{photo}$, $\mathcal{L}_s(c_{5,10}) = \texttt{upload}$, *and* $\mathcal{L}_s(c_{9,7}) = \texttt{upload}$. *In Fig. 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 Fig. 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 Fig. 1, cells with locally detected dynamic requests* `extinguish` *and* `assist` *are shown in red and cyan, respectively.*

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$_{-\mathbf{X}}$ formula ϕ_g (Sec. II) over the set of static requests $\mathcal{S}$ and dictates the global motion of the vehicle in the environment. The local mission specification is concerned with the dynamic requests detected locally within the sensing range of the vehicle and specifies how the vehicle must respond to them. Specifically, the local mission specification comprises a set of *service* requests $\mathcal{D}_{service}$ and a set of *avoidance* requests $\mathcal{D}_{avoid}$ such that $\mathcal{D} = \mathcal{D}_{service} \cup \mathcal{D}_{avoid}$ and $\mathcal{D}_{service} \cap \mathcal{D}_{avoid} = \emptyset$, where $\mathcal{D}$ is the set of dynamic requests from (1). If the vehicle locally detects a dynamic service request from the set $\mathcal{D}_{service}$, it must service the request by going to the cell, or cells, associated with the request. If the vehicle locally detects a dynamic avoidance request from the set $\mathcal{D}_{avoid}$, it must avoid the cell, or cells, associated with the request. In case of multiple local requests, the order in which the vehicle handles the service requests is determined by their priority values given by a priority function $prio : \mathcal{D}_{service} \to \mathbb{N}$ with lower values meaning higher priority. Avoidance requests, on the other hand, should always be satisfied by the vehicle and do not have associated priority values. Note that as the vehicle moves in the environment, the static and dynamic requests it satisfies produce a word over $\mathcal{S} \cup \mathcal{D}$ (Sec. II) which can be checked against the global and local mission specifications. We can now formulate the problem that we consider in this paper.

Problem III.2. *Given an environment* $\mathcal{E}$, *a global mission specification* ϕ_g *in the form of an LTL*$_{-\mathbf{X}}$ *formula over* $\mathcal{S}$, *and a local mission specification* $\mathcal{D} = \mathcal{D}_{service} \cup \mathcal{D}_{avoid}$, $prio : \mathcal{D} \to \mathbb{N}$, *find a vehicle control strategy such that the produced trajectory satisfies both the global and the local mission specifications.*

Example III.1 Revisited. *The quadrotor given in Fig 1 is required to complete a persistent surveillance mission starting at* $c_{3,1}$. *The global mission specification is to*

> *Keep taking photos and upload current photo before taking another photo.*

This can be expressed in LTL$_{-\mathbf{X}}$ *as*

$$\phi_g := \square\diamond \texttt{photo} \wedge \square(\texttt{photo} \Rightarrow (\texttt{photo}\, \mathcal{U}\, (\neg\texttt{photo}\, \mathcal{U}\, \texttt{upload}))). \quad (2)$$

The local mission specification is defined such that $\mathcal{D}_{service} = \{\texttt{extinguish}, \texttt{assist}\}$, $\mathcal{D}_{avoid} = \{\texttt{unsafe}\}$, *with* $prio(\texttt{assist}) = 0$ *and* $prio(\texttt{extinguish}) = 1$, *i.e. if both fire and survivor are detected locally, priority is given to the survivor.*

Our solution to Prob. III.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 vehicle visits the cells with static requests satisfies the global mission specification ϕ_g, and
- the local motion of the vehicle within its sensing range satisfies the local mission specification.

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 (Sec. IV-A). Then, each time the vehicle arrives at a cell, our controller translates the local mission specification to an scLTL formula ϕ_l and obtains the corresponding FSA (Def. II.3). Following this, a local product automaton $\mathbf{L}$ that captures the local information obtained from the sensors, the motion of the vehicle between the cells, and ϕ_l is constructed (Sec. IV-B). Next, our controller uses $\mathbf{L}$ and $\mathbf{G}$ to compute the next cell the vehicle must go to so that ϕ_g and ϕ_l are satisfied (Sec. IV-C). 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 Exp. III.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 (Sec. V).

Remark III.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 [9] for unicycles and Sec. V-B 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 [3]. Note that non-polytopic regions resulting from other types of partitioning schemes can, in principle, be under- or over-approximated by polytopic regions.*

IV. Problem Solution

A. Global Product Automaton

The first step in our solution is to construct a global product automaton $\mathbf{G}$ that captures the global mission specification ϕ_g and the motion of the vehicle among the cells with static requests. This automaton is constructed only once at the beginning of a run by Alg. 3 and is used for driving the vehicle between the cells with static requests so that ϕ_g is satisfied. To this end, we first construct a transition system $\mathbf{T} := (\mathcal{Q}_\mathrm{T}, q_{\mathrm{T},init}, \delta_\mathrm{T}, \Pi_\mathrm{T}, h_\mathrm{T}, w_\mathrm{T})$ (Def. II.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. We define the set of states of $\mathbf{T}$ as $\mathcal{Q}_\mathrm{T} = \{c \,|\, c \in \mathcal{C}, \mathcal{L}_s(c) \text{ is defined}\} \cup \{c_{init}\}$, i.e. we have a state for each cell with a static request and the initial cell of the vehicle. Then, we define $\Pi_\mathrm{T} = \mathcal{S}$ and set $h_\mathrm{T}(q) = \mathcal{L}_s(q)$ for all $q \in \mathcal{Q}_\mathrm{T}$. Next, we define the transitions (edges) between the states of $\mathbf{T}$ such that the weight $w_\mathrm{T}(q, q')$ of the transition between $q, q' \in \mathcal{Q}_\mathrm{T}$ is the length of 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 Sec. V), we use Manhattan distance to calculate w_T. Each state $q \in \mathcal{Q}_\mathrm{T}$ has a self loop so that the vehicle can stay at a given cell (corresponding continuous controllers are described in Sec. V). We set the weights of these loops to be ϵ, where $0 < \epsilon << 1$, instead of zero so that the movement of the vehicle in the environment is not blocked due to zero weight self-loops as we discuss in Sec. IV-C.

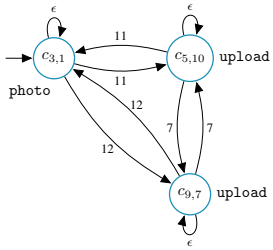

Fig. 2. Transition system $\mathbf{T}$ modeling the motion of the quadrotor among the cells with static requests for the environment given in Fig. 1. Static requests are shown next to their corresponding states and $q_{\mathrm{T},init} = c_{3,1}$.

Example III.1 Revisited. *Fig. 2 illustrates the transition system* $\mathbf{T}$ *modeling the motion of the quadrotor between the cells with static requests* `photo` *and* `upload` *for the environment given in Fig. 1.*

Then, we obtain the Büchi automaton $\mathbf{B}$ that corresponds to the global mission specification ϕ_g and construct the product automaton $\mathbf{G} := \mathbf{T} \otimes \mathbf{B}$ as defined next.

Definition IV.1 (Product of T and B). *The product of a weighted transition system* $\mathbf{T} := (\mathcal{Q}_\mathrm{T}, q_{\mathrm{T},init}, \delta_\mathrm{T}, \Pi_\mathrm{T}, h_\mathrm{T}, w_\mathrm{T})$ *(Def. II.1) and a Büchi automaton* $\mathbf{B} := (\mathcal{Q}_\mathrm{B}, \mathcal{Q}_{\mathrm{B},init}, \delta_\mathrm{B}, \Sigma_\mathrm{B}, \mathcal{F}_\mathrm{B})$ *(Def. II.2) is a tuple* $\mathbf{G} := (\mathcal{Q}_\mathrm{G}, \mathcal{Q}_{\mathrm{G},init}, \delta_\mathrm{G}, w_\mathrm{G}, \mathcal{F}_\mathrm{G})$*, where*

- $\mathcal{Q}_\mathrm{G} \subseteq \mathcal{Q}_\mathrm{T} \times \mathcal{Q}_\mathrm{B}$ *is the finite set of states that are reachable from* $\mathcal{Q}_{\mathrm{G},init}$*;*
- $\mathcal{Q}_{\mathrm{G},init} = \{(q_{\mathrm{T},init}, q'_\mathrm{B}) \,|\, (q_\mathrm{B}, h_\mathrm{T}(q_{\mathrm{T},init}), q'_\mathrm{B}) \in \delta_\mathrm{B} \;\forall\; q_\mathrm{B} \in \mathcal{Q}_{\mathrm{B},init}\}$ *is the set of initial states;*
- $\delta_\mathrm{G} = \{((q_\mathrm{T}, q_\mathrm{B}), (q'_\mathrm{T}, q'_\mathrm{B})) \,|\, (q_\mathrm{T}, q'_\mathrm{T}) \in \delta_\mathrm{T},\ (q_\mathrm{B}, h_\mathrm{T}(q'_\mathrm{T}), q'_\mathrm{B}) \in \delta_\mathrm{B}\}$ *is the transition relation;*
- $w_\mathrm{G}((q_\mathrm{T}, q_\mathrm{B}), (q'_\mathrm{T}, q'_\mathrm{B})) = w_\mathrm{T}(q_\mathrm{T}, q'_\mathrm{T})$ *for all* $((q_\mathrm{T}, q_\mathrm{B}), (q'_\mathrm{T}, q'_\mathrm{B})) \in \delta_\mathrm{G}$ *is the weight function;*
- $\mathcal{F}_\mathrm{G} = \{(q_\mathrm{T}, q_\mathrm{B}) \,|\, (q_\mathrm{T}, q_\mathrm{B}) \in \mathcal{Q}_\mathrm{G}, q_\mathrm{B} \in \mathcal{F}_\mathrm{B}\}$ *is the set of accepting states.*

Note that the global product automaton $\mathbf{G}$ captures both the motion of the vehicle between the cells with static requests and the global mission specification, hence the name *global* product automaton. We discuss how we use $\mathbf{G}$ to enforce the satisfaction of the global mission specification ϕ_g in Sec. IV-C.

B. Local Product Automaton

The local product automaton $\mathbf{L}$ that we discuss in this section is constructed at each iteration of Alg. 3, i.e. each time the vehicle arrives at a cell, using the sensors of the vehicle.

To satisfy ϕ_g, Alg. 3 drives the vehicle between the cells of $\mathcal{Q}_\mathrm{T}$, and for each pair of cells c_{cur} and c_{next} that it drives the vehicle between, it uses $\mathbf{L}$ to find local plans that satisfy the local mission specification and get the vehicle closer to c_{next}.

Algorithm 1: Construct the local transition system $\mathbf{U}$.

Input: The cells $c_{cur}, c_{next} \in \mathcal{Q}_\mathrm{T}$ between which the vehicle is moving and x, y coordinates of the current cell of the vehicle.
Output: Local transition system $\mathbf{U}$.

1 $\mathcal{Q}_\mathrm{U} = \{c_{i,j} : |x-i| \leq \frac{o-1}{2}, |y-j| \leq \frac{p-1}{2}, c_{i,j} \in \mathcal{C}\}$.
2 $q_{\mathrm{U},init} = c_{x,y}$.
3 Set $\mathcal{Q}_\mathrm{U} = \mathcal{Q}_\mathrm{U} \setminus (\mathcal{Q}_\mathrm{T} \setminus \{c_{next}\})$.
4 **if** $c_{x,y} = c_{cur}$ **then** Add c_{cur} back to $\mathcal{Q}_\mathrm{U}$.
5 $\delta_\mathrm{U} = \{(c_{i,j}, c_{k,l}) \,|\, c_{i,j}, c_{k,l} \in \mathcal{Q}_\mathrm{U}, (|k-i| = 1 \wedge l = j) \vee (|l-j| = 1 \wedge i = k)\}$.
6 $w_\mathrm{U}(c_{i,j}, c_{k,l}) = 1$ for all $(c_{i,j}, c_{k,l}) \in \delta_\mathrm{U}$.
7 $h_\mathrm{U}(c_{i,j}) = \mathcal{L}_d(t)(c_{i,j})$ for $c_{i,j} \in \mathcal{Q}_\mathrm{U}$.
8 Add end to $\mathcal{Q}_\mathrm{U}$ with $h_\mathrm{U}(end) = end$.
9 **if** $c_{next} \in \mathcal{Q}_\mathrm{U}$ **then** Add (c_{next}, end) to δ_U.
10 **else**
11 Add (c_{bndry}, end) to δ_U for each cell c_{bndy} at the boundary of the sensing range.
12 Π_U is the set of propositions satisfied at the states of $\mathbf{U}$.
13 **return** $\mathbf{U} := (\mathcal{Q}_\mathrm{U}, q_{\mathrm{U},init}, \delta_\mathrm{U}, \Pi_\mathrm{U}, h_\mathrm{U}, w_\mathrm{U})$.

We use Alg. 1 to construct the local transition system $\mathbf{U}$ that models the motion of the vehicle between the cells within its sensing range and captures the dynamic requests sensed at each cell. In lines 1-2, we initialize the set of states of $\mathbf{U}$ as an $o \times p$ grid of square cells centered at the current cell $c_{x,y}$ of the vehicle (the sensing grid discussed in Sec. III). To be able to guarantee correctness, we need to make sure that as the vehicle moves between cells $c_{cur}, c_{next} \in \mathcal{Q}_\mathrm{T}$, it does not visit any cells in $\mathcal{Q}_\mathrm{T}$ other than c_{next} once it leaves c_{cur}. To this end, we first remove all cells in $\mathcal{Q}_\mathrm{U}$ that are also in $\mathcal{Q}_\mathrm{T}$ except c_{next} (line 3). If the vehicle has not left c_{cur} yet, we add it back to $\mathcal{Q}_\mathrm{U}$ as required (line 4). In lines 5-7, we define unit weight transitions between all adjacent cells in $\mathcal{Q}_\mathrm{U}$ and assign locally detected dynamic requests to their corresponding cells. Then, in lines 8-11, we add a blocking *end* state to $\mathcal{Q}_\mathrm{U}$ and define necessary transitions from those states where the controller given in Sec. IV-C may end its local planning at each iteration. If $c_{next} \in \mathcal{Q}_\mathrm{U}$, we add (c_{next}, end) to δ_U as it is the current target cell of the vehicle. Otherwise, we add a transition from all boundary cells of the sensing grid of the vehicle to the *end* state to allow the controller to plan until the end of the sensing range.

Then, using Alg. 2, we translate the local mission specification (Sec. III) to the scLTL formula ϕ_l (Sec. II). Note that Alg. 2 omits the parts of the local mission specification that do not apply to the current set of dynamic requests sensed by the vehicle (line 3). Finally, we obtain the deterministic FSA $\mathbf{A}$ (Def. II.3) that corresponds to ϕ_l and construct the local product automaton $\mathbf{L} := \mathbf{U} \otimes \mathbf{A}$ as defined next.

Algorithm 2: Obtain scLTL formula ϕ_l.

Input: Local mission specification as defined in Sec. III and transition system $\mathbf{U}$.
Output: The corresponding scLTL formula ϕ_l.

1 service_set = $\emptyset$, ϕ_l = (True $\mathcal{U}$ end).
2 **foreach** $request \in \mathcal{D}_{service} \cup \mathcal{D}_{avoid}$ **do**
3 **if** $request \notin \Pi_U$ **then** Continue.
4 **if** $request \in \mathcal{D}_{service}$ **then**
5 $service\_set = service\_set \cup \{request\}$.
6 Append '$\wedge$ ($\diamond$ $request$)' to ϕ_l.
7 **else**
8 Append '$\wedge$ ($\neg request\, \mathcal{U}\, end$)' to ϕ_l.
9 **foreach** *(req1, req2)* $\in$ *service_set* $\times$ *service_set* **do**
10 **if** *req1* $\neq$ *req2 and prio(req1)* $<$ *prio(req2)* **then**
11 Append '$\wedge$ ($\neg$ req2 $\mathcal{U}$ req1)' to ϕ_l.
12 **return** ϕ_l.

Definition IV.2 (Product of U and A). *The product of a weighted transition system* $\mathbf{U} := (\mathcal{Q}_\mathrm{U}, q_{\mathrm{U},init}, \delta_\mathrm{U}, \Pi_\mathrm{U}, h_\mathrm{U}, w_\mathrm{U})$ *(Def. II.1) and a deterministic finite state automaton* $\mathbf{A} := (\mathcal{Q}_\mathrm{A}, q_{\mathrm{A},init}, \delta_\mathrm{A}, \Sigma_\mathrm{A}, \mathcal{F}_\mathrm{A})$ *(Def. II.3) is a tuple* $\mathbf{L} := (\mathcal{Q}_\mathrm{L}, q_{\mathrm{L},init}, \delta_\mathrm{L}, w_\mathrm{L}, \mathcal{F}_\mathrm{L})$*, where*

- $\mathcal{Q}_\mathrm{L} \subseteq \mathcal{Q}_\mathrm{U} \times \mathcal{Q}_\mathrm{A}$ *is the finite set of states that are reachable from the initial state;*
- $q_{\mathrm{L},init} = (q_{\mathrm{U},init}, q'_\mathrm{A})$ *is the initial state such that* $(q_{\mathrm{A},init}, h_\mathrm{U}(q_{\mathrm{U},init}), q'_\mathrm{A}) \in \delta_\mathrm{A}$*;*
- $\delta_\mathrm{L} = \{((q_\mathrm{U}, q_\mathrm{A}), (q'_\mathrm{U}, q'_\mathrm{A})) \,|\, (q_\mathrm{U}, q'_\mathrm{U}) \in \delta_\mathrm{U},\ (q_\mathrm{A}, h_\mathrm{U}(q'_\mathrm{U}), q'_\mathrm{A}) \in \delta_\mathrm{A}\}$ *is the transition relation;*
- $w_\mathrm{L}((q_\mathrm{U}, q_\mathrm{A}), (q'_\mathrm{U}, q'_\mathrm{A})) = w_\mathrm{U}(q_\mathrm{U}, q'_\mathrm{U})$ *for all* $((q_\mathrm{U}, q_\mathrm{A}), (q'_\mathrm{U}, q'_\mathrm{A})) \in \delta_\mathrm{L}$ *is the weight function;*
- $\mathcal{F}_\mathrm{L} = \{(q_\mathrm{U}, q_\mathrm{A}) \,|\, (q_\mathrm{U}, q_\mathrm{A}) \in \mathcal{Q}_\mathrm{L}, q_\mathrm{A} \in \mathcal{F}_\mathrm{A}\}$ *is the set of accepting states.*

Note that the local product automaton $\mathbf{L}$ obtained as the product of $\mathbf{U}$ and $\mathbf{A}$ captures both the motion of the vehicle within its local sensing range and the local mission specification ϕ_l, hence the name *local*. Next, we show how we use the local product automaton $\mathbf{L}$ and the global product automaton $\mathbf{G}$ from Sec. IV-A to control the vehicle.

C. Receding Horizon Controller

The controller that we propose as a solution to Prob. III.2 is presented in the form of Alg. 3. In the following, we discuss each step of Alg. 3 in detail.

Lines 1–6 of Alg. 3 are run only once and are responsible for the initialization of the data structures related to the global part of the mission. In line 1 of Alg. 3, we construct the global product automaton $\mathbf{G}$ as the product of the transition system $\mathbf{T}$ that models the behavior of the vehicle between the cells with static requests and the Büchi automaton $\mathbf{B}$ that captures the global mission specification ϕ_g as discussed in Sec. IV-A. Note that the acceptance condition for a Büchi automaton is to visit an accepting state infinitely often (Sec. II). Thus, in

Algorithm 3: Vehicle Controller.

Input: Environment $\mathcal{E}$, global mission specification ϕ_g, local mission specification $\mathcal{D} = \mathcal{D}_{service} \cup \mathcal{D}_{avoid}, prio : \mathcal{D} \to \mathbb{N}$.

Output: Next grid cell to fly to.

Executed Offline (initialization part):

1 Construct the global product automaton $\mathbf{G}$ (Def. IV.1).
2 Remove all those states in $\mathcal{F}_{\mathrm{G}}$ that cannot reach themselves.
3 **if** $\mathcal{F}_{\mathrm{G}} = \emptyset$ **then**
4 Abort: ϕ_g cannot be satisfied.
5 Let $fd(q) = \min_{q' \in \mathcal{F}_{\mathrm{G}}} shortest\_dist(q, q')$ for all $q \in \mathcal{Q}_{\mathrm{G}}$.
6 $g_{cur} = \arg\min_{q \in \mathcal{Q}_{init}} fd(q)$.

Executed Online (receding horizon part):

7 **while** *True* **do**
8 Obtain the local scLTL formula ϕ_l using Alg. 2.
9 Construct the FSA $\mathbf{A}$ corresponding to ϕ_l (Sec. IV-B).
10 $d^\star = \infty$.
11 **foreach** $g_{next} \in \{g_{next} \mid (g_{cur}, g_{next}) \in \delta_{\mathrm{G}}\}$ **do**
12 Construct $\mathbf{U}$ for $(g_{cur}[0], g_{next}[0])$ using Alg. 1.
13 Construct the local product automaton $\mathbf{L} := \mathbf{U} \times \mathbf{A}$ (Def. IV.2).
14 **foreach** $q \in \mathcal{F}_{\mathrm{L}}$ **do**
15 $d_{cell} = man\_dist(q[0], g_{next}[0])$.
16 $d_{plan} = d_{cell} + fd(g_{next})$.
17 **if** $d_{plan} < d^\star$ **then**
18 $d^\star = d_{plan}$.
19 $cell^\star_{next}$ = Next cell on shortest path from $q_{\mathrm{L},init}$ to q.
20 $g^\star_{next} = g_{next}$.
21 **if** $d^\star = \infty$ **then**
22 Abort: No feasible local plan.
23 **if** $cell^\star_{next}$ *is* $g^\star_{next}[0]$ **then** $g_{cur} = g^\star_{next}$.
24 Apply controls to reach $\mathrm{cell}^\star_{\mathrm{next}}$.

line 2 of Alg. 3 we remove all those accepting states in $\mathcal{F}_{\mathrm{G}}$ of $\mathbf{G}$ that cannot reach themselves. In lines 3-4 of Alg. 3, we abort if there are no accepting states left in $\mathcal{F}_{\mathrm{G}}$, meaning that the global mission specification ϕ_g cannot be satisfied. In line 5, we define a potential-like function $fd(q)$ that returns the shortest distance to the set of accepting states $\mathcal{F}_{\mathrm{G}}$ for a state $q \in \mathcal{Q}_{\mathrm{G}}$. The $shortest\_dist(q, q')$ function that we use here returns the the distance of the shortest path between states $q, q' \in \mathcal{Q}_{\mathrm{G}}$ using Dijkstra's shortest path algorithm [10]. Note that the value returned by $fd(q)$ decreases as we get closer to the set of accepting states $\mathcal{F}_{\mathrm{G}}$ and it returns zero once we are at an accepting state $q \in \mathcal{F}_{\mathrm{G}}$. For a state $q \in \mathcal{Q}_{\mathrm{G}}$ that cannot reach any accepting state in $\mathcal{F}_{\mathrm{G}}$, $fd(q) = \infty$. As the global product automaton $\mathbf{G}$ may have multiple initial states due to the nondeterminism of $\mathbf{B}$, in line 8 we set our current state in $\mathbf{G}$, denoted by g_{cur}, to the state in $\mathcal{Q}_{\mathrm{G},init}$ that is closest to the set of accepting states $\mathcal{F}_{\mathrm{G}}$.

The rest of Alg. 3 (lines 7–24) is an infinite loop that executes once at every new cell reached by the vehicle. In this part, our controller locally plans a path that satisfies the local mission specification while taking the vehicle as close as possible towards satisfaction of the global mission specification. Thus, it is essentially this second part of Alg. 3 which makes our controller a receding horizon controller. We proceed by obtaining the scLTL formula ϕ_l corresponding to the parts of the local mission specification that applies to the current requests sensed by the vehicle using Alg. 2. Then at line 9, we construct the FSA $\mathbf{A}$ that corresponds to ϕ_l.

The loop in lines 11–20 picks the best local trajectory within the local sensing range of the vehicle such that it both satisfies ϕ_l and takes the vehicle closest to satisfaction of the global mission specification. To this end, we consider all neighbors, denoted by g_{next}, of our current state g_{cur} in $\mathbf{G}$ (line 11). For each g_{next}, we construct the local transition system $\mathbf{U}$ that models the motion of the vehicle within its local sensing range using Alg. 1 (line 12). Note that $\mathbf{U}$ excludes all cells from $\mathcal{Q}_{\mathrm{T}}$ except $g_{next}[0]$ (and $g_{cur}[0]$ if the vehicle has not left that cell yet), where $g_{cur}[0]$ and $g_{next}[0]$ are the cells corresponding to the states g_{cur} and g_{next} in the global product automaton $\mathbf{G}$, respectively. This guarantees that the vehicle will not satisfy any unintended static requests that can potentially violate the global mission specification as it goes from $g_{cur}[0]$ to $g_{next}[0]$ in the environment. In line 13, we construct the local product automaton $\mathbf{L}$ as the product of $\mathbf{U}$ and $\mathbf{A}$.

In lines 14–20 we iterate over all possible states in $\mathbf{L}$ where a satisfying local plan can end. For each accepting state $q \in \mathcal{F}_{\mathrm{L}}$, we set d_{cell} to the Manhattan distance between the cell corresponding to q, denoted by $q[0]$, and the cell corresponding to g_{next}, denoted by $g_{next}[0]$ (line 15). Here, $man\_dist()$ returns the Manhattan distance between the cells corresponding $q[0]$ and $g_{next}[0]$. Then, in line 16, we calculate the predicted distance of the vehicle to the accepting states of $\mathbf{G}$ after reaching $q[0]$ as the sum of d_{cell} and $fd(g_{next})$. Lines 17–20 keep track of the next cell, denoted by $cell^\star_{next}$, of the best local plan obtained so far corresponding to the combination of $g_{next} \in \mathcal{Q}_{\mathrm{G}}$ and $q \in \mathcal{F}_{\mathrm{L}}$ that takes the vehicle closest to the set of accepting states of $\mathbf{G}$. In line 21, we abort if we cannot find a feasible local plan that both satisfies ϕ_l and flies the vehicle towards some neighbor g_{next} of g_{cur} that can reach $\mathcal{F}_{\mathrm{G}}$. Else, in lines 23–24, we update g_{cur} as necessary and fly the vehicle to $cell^\star_{next}$. Once the vehicle reaches $cell^\star_{next}$, Alg. 3 continues execution from line 7. Next, we show the correctness of Alg. 3.

Theorem IV.3. *Assume that during its execution, Alg. 3 reaches line 23 with $fd(g^\star_{next}) = 0$ infinitely often. Then, the trajectory generated by the vehicle satisfies the global and local mission specifications.*

Proof: Alg. 3 guides the vehicle between the cells with static requests by computing local plans. Under the given

assumption, the controller satisfies the Büchi acceptance condition (Sec. II) by visiting the set of accepting states of $\mathbf{G}$ infinitely often. Since we are also guaranteed not to visit any cells with static requests besides $g_{next}[0]$ due to the way $\mathbf{U}$ is constructed (Sec. IV-B), the motion of the vehicle in the environment satisfies the global mission specification ϕ_g. Note also that any local plan computed using Alg. 3 is guaranteed to satisfy the local mission specification as it ends at an accepting state of the local product automaton $\mathbf{L}$. Thus, under the given assumption, Alg. 3 correctly solves Prob. III.2. ■

Remark IV.4. *The assumption employed in Thm. IV.3 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 (or equivalently there is no local mission specification) and the cells with static requests are at least one cell apart. In this case, if the global mission ϕ_g is satisfiable, then Alg. 3 is guaranteed to satisfy it.*

V. Implementation and Experimental Results

A. Software Implementation and Experimental Setup

The controller presented in Alg. 3 is implemented as a Python module that returns the next cell the quadrotor must fly to given its current location (provided by the motion capture system) and current sensor measurements (provided by the Matlab code that processes images sent by the quadrotor). In our implementation, we use ltl2ba [7] to obtain the Büchi automaton $\mathbf{B}$ corresponding to ϕ_g, and use scheck [14] to obtain the FSA $\mathbf{A}$ corresponding to ϕ_l (Alg. 3). The low-level controllers that control the flight of the quadrotor from one grid cell to another according to the output of Alg. 3 are implemented in Matlab. We discuss these low-level quadrotor controllers in Sec. V-B.

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 (a 5×5 cell grid). The colors used in the experiments are as given in Exp. III.1 and illustrated in Fig. 1: green for `photo`, blue for `upload`, yellow for `unsafe`, red for `extinguish`, and cyan for `assist`. The computers run the code responsible for trajectory planning (Alg. 3) and low-level control of the quadrotor as well as processing the images sent by the quadrotor.

B. Quadrotor Low Level Controller

The quadrotor low level controllers that ensure the safe and smooth transition from one cell to another (as determined by Alg. 3) are based on (1) input-output linearization for the quadrotor dynamics [16], and (2) construction of multi-affine vector fields for control-to-facet and invariance in rectangles [2, 9]. Due to space limitations, the details are omitted. In short, 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 Alg. 3, then remains in the target cell while performing the necessary sensing and planning operations. 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 [2, 9]. Fig. 3 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 controllers are designed to use the absolute position of the quadrotor to define the resulting velocity reference for the feedback loop. 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 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 dynamics equations for an orientation and thrust that produces the appropriate motion, while accounting for the non-linear dynamics in the model [16]. The desired attitude is then used as a reference for the inner control loop, designed to stabilize the attitude. 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 Fig. 3)

C. Experimental Results

In this section, we return to the persistent surveillance mission given in Exp. III.1 and present the results of our experiments where a quadrotor satisfies the global and local mission specifications given in Exp. III.1. Fig. 3 illustrates the trajectory followed by the quadrotor for a particular realization of dynamic requests during the experiments. Here, the black line corresponds to the actual flight path of the quadrotor over the environment as provided by the motion capture system. At the beginning of the flight, only the `unsafe` dynamic avoidance request is there. The remaining dynamic service requests `extinguish` and `assist` occur later in the flight as we discuss next. The quadrotor begins its flight at $c_{3,1}$

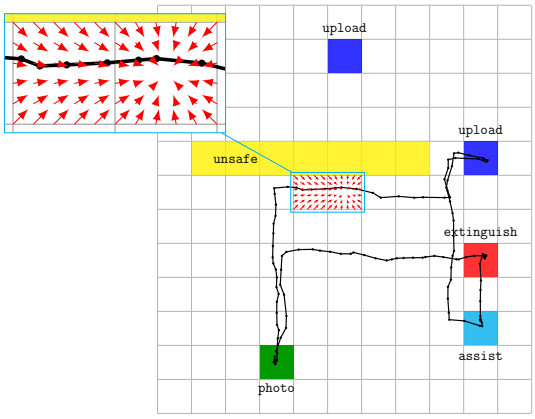

Fig. 3. Quadrotor trajectory plotted over the environment in Exp. III.1. The closeup on cells $c_{4,6}$ and $c_{5,6}$ show the vector fields computed for flying the quadrotor from $c_{4,6}$ to $c_{5,6}$.

servicing the static `photo` request. Next, the quadrotor has to service the static `upload` request at either $c_{5,10}$ or $c_{9,7}$. As the quadrotor cannot detect the unsafe cells yet and $c_{5,10}$ is closer to its current position than $c_{9,7}$, it starts flying towards $c_{5,10}$. Once the quadrotor reaches $c_{3,6}$ it can no longer go north due to the unsafe cells and can only fly east to get closer to $c_{5,10}$. However, when the quadrotor arrives at $c_{5,6}$, the controller finds that flying towards $c_{9,7}$ takes the quadrotor closer to satisfying ϕ_g than flying to $c_{5,10}$ does, so the quadrotor starts flying towards $c_{9,7}$ to service the `upload` request. After reaching $c_{9,7}$, the quadrotor needs to fly back to $c_{3,1}$ to service the `photo` request as required by ϕ_g. As the quadrotor leaves $c_{9,7}$ for $c_{3,1}$, the `extinguish` and `assist` dynamic requests appear at $c_{9,4}$ and $c_{9,2}$, respectively. However, due to its limited sensing range, the quadrotor only detects the `extinguish` request and starts flying south to reach $c_{9,4}$. At $c_{8,4}$, the quadrotor detects the `assist` request and flies to $c_{9,2}$ as assisting a survivor is of higher priority than extinguishing a fire according to the local mission specification. After assisting the survivor at $c_{9,2}$, the quadrotor extinguishes the fire at $c_{9,4}$ and flies to $c_{3,1}$ to service the `photo` request. Note that, 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 Fig. 3 is a portion of the infinite mission of the quadrotor. Our video submission accompanying the paper shows the actual flight of the quadrotor in our experimental setup. During the experiments, worst case execution times of the offline and online portions of Alg. 3 were 25 ms and 180 ms, respectively, when executed on an iMac i5 quad-core computer. Total flight time of the quadrotor for the trajectory shown in Fig. 3 was ≈ 90 seconds.

VI. Conclusion

We 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 local requests sensed locally are serviced as well. Our receding horizon implementation of controllers insures fast responses to rapidly changing local requests. We demonstrated the applicability of our approach with experiments involving a quadrotor performing a persistent surveillance task over a planar grid environment.

Acknowledgments

This work was partially supported by the ONR under grants MURI N00014-09-1051 and MURI N00014-10-10952 and by NSF under grant CNS-1035588.

References

[1] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[2] C. Belta and L. C. G. J. M. Habets. Control of a class of nonlinear systems on rectangles. *IEEE Transactions on Automatic Control*, 51 (11):1749 – 1759, 2006.

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

[4] A. Bhatia, L. E. Kavraki, and M.Y. Vardi. Sampling-based motion planning with temporal goals. In *IEEE Intl. Conf. Robotics and Automation*, pages 2689–2696, may 2010.

[5] X. C. Ding, M. Kloetzer, Y. Chen, and C. Belta. Formal methods for automatic deployment of robotic teams. *IEEE Robotics & Automation Magazine*, 18:75–86, 2011.

[6] X. C. Ding, M. Lazar, and C. Belta. Receding horizon temporal logic control for finite deterministic systems. In *American Control Conference*, Montreal, Canada, 2012.

[7] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, 2001. Springer.

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

[9] M. Kloetzer and C. Belta. A framework for automatic deployment of robots in 2d and 3d environments. In *IEEE/RSJ Intl. Conf. Intelligent Robots & Systems*, pages 953–958, 2006.

[10] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms, Vol. 21 of Algorithmics and Combinatorics*. Springer, 4th edition, 2007.

[11] H. Kress-Gazit, G. Fainekos, and G. J. Pappas. Where's waldo? sensor-based temporal logic motion planning. In *IEEE Intl. Conf. Robotics and Automation*, pages 3116–3121, 2007.

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

[13] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19:291–314, October 2001.

[14] T. Latvala. Efficient model checking of safety properties. In *Model Checking Software. 10th International SPIN Workshop*, pages 74–88. Springer, 2003.

[15] D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63 (5):243–246, 1997.

[16] H. Voos. Nonlinear control of a quadrotor micro-uav using feedback-linearization. In *IEEE Intl. Conf. Mechatronics*, pages 1–6, 2009.

[17] T. Wongpiromsarn, U. Topcu, and R. M. Murray. Receding horizon control for temporal logic specifications. In *Hybrid systems: Computation and Control*, pages 101–110, Stockholm, Sweden, 2010.