Article

The International Journal of Robotics Research
2020, Vol. 39(8) 1002–1028

DOI: 10.1177/0278364920918919
journals.sagepub.com/home/ijr
SAGE

Reactive sampling-based path planning with temporal logic specifications

Cristian Ioan Vasile[1], Xiao Li[2] and Calin Belta[2]

Abstract

We develop a sampling-based motion planning algorithm that combines long-term temporal logic goals with short-term reactive requirements. The mission specification has two parts: (1) a global specification given as a linear temporal logic (LTL) formula over a set of static service requests that occur at the regions of a known environment, and (2) a local specification that requires servicing a set of dynamic requests that can be sensed locally during the execution. The proposed computational framework consists of two main ingredients: (a) an off-line sampling-based algorithm for the construction of a global transition system that contains a path satisfying the LTL formula; and (b) an on-line sampling-based algorithm to generate paths that service the local requests, while making sure that the satisfaction of the global specification is not affected. The off-line algorithm has four main features. First, it is incremental, in the sense that the procedure for finding a satisfying path at each iteration scales only with the number of new samples generated at that iteration. Second, the underlying graph is sparse, which implies low complexity for the overall method. Third, it is probabilistically complete. Fourth, under some mild assumptions, it has the best possible complexity bound. The on-line algorithm leverages ideas from LTL monitoring and potential functions to ensure progress towards the satisfaction of the global specification while servicing locally sensed requests. Examples and experimental trials illustrating the usefulness and the performance of the framework are included.

Keywords

Sampling-based planning, linear temporal logic, reactive planning

1. Introduction

Motion planning is a fundamental problem in robotics (LaValle, 2006). The goal is to generate a feasible path for a robot to move from an initial to a final configuration while avoiding obstacles. Approaches based on potential fields, navigation functions, and cell decompositions are among the most commonly used (Choset et al., 2005). These, however, become prohibitively expensive in high-dimensional configuration spaces. Sampling-based methods were proposed to overcome this limitation. Examples include the probabilistic roadmap (PRM) algorithm proposed by Kavraki et al. (1996), which is very useful for multi-query problems, but is not well suited for the integration of differential constraints. LaValle and Kuffner (2001) proposed rapidly-exploring random trees (RRT). These grow randomly, are biased to explore "new" space (LaValle and Kuffner, 2001) (Voronoi bias), and find solutions quite fast. Moreover, PRM and RRT were shown to be probabilistically complete (Kavraki et al., 1996; LaValle and Kuffner, 2001), but not probabilistically optimal (Karaman and Frazzoli, 2011). Karaman and Frazzoli (2011) proposed RRT* and PRM*, the probabilistically optimal counterparts of RRT and PRM. Sampling-based methods have been employed in dealing with kinematic constraints (Hauser and Zhou, 2016; Kleinbort et al., 2019; Moore et al., 2014; Webb and van den Berg, 2013), stochastic robot models (Agha-mohammadi et al., 2014; Burns and Brock, 2007; Hauser, 2011; van den Berg et al., 2011; Vasile et al., 2016), multi-robot systems (Dobson et al., 2017; Kantaros and Zavlanos, 2019), in applications such as autonomous driving (Kuwata et al., 2009; Reyes Castro

[1]Laboratory for Information and Decision Systems,Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA
[2]Department of Mechanical Engineering, Boston University, Boston, MA, USA

Corresponding author:
Cristian-Ioan Vasile, Laboratory for Information and Decision Systems, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 77 Massachusetts Avenue, 32-D716, Cambridge, MA 02139, USA.
Email: cvasile@mit.edu

et al., 2013; Vasile et al., 2017b), manipulation (He et al., 2017; Muhayyuddin et al., 2018), and surgery (Baykal et al., 2019). A more detailed exposition of sampling-based methods can be found in Kingston et al. (2018).

A recent trend in robot motion planning is the development of computational frameworks that allow for automatic deployment from rich, high-level, temporal logic specifications. As opposed to traditional methods, which only allow to specify a goal position, these frameworks can capture more complex tasks such as sequencing (e.g., "Reach *A*, then *B*, and then *C*"), convergence ("Go to *A* and stay there for all future times"), persistent surveillance ("Visit *A*, *B*, and *C*, in this order, infinitely often"), and more complex logical combinations of the above, such as "Visit *A* and then *B* or *C* infinitely often. Always avoid *D*. Never go to *E* unless *F* was reached before." It was shown that temporal logics, such as linear temporal logic (LTL), computational tree logic (CTL), and μ-calculus, and their probabilistic versions (PLTL, PCTL), can be used as formal languages for motion planning (Bhatia et al., 2010; Ding et al., 2011; Karaman and Frazzoli, 2009; Kress-Gazit et al., 2007, 2018; Nilsson et al., 2016; Schillinger et al., 2018; Wongpiromsarn et al., 2009). Adapted model checking algorithms and automata game techniques (Belta et al., 2017; Chen et al., 2012; Kress-Gazit et al., 2007) have been used to generate plans and control policies for finite models of robot motion. Such models were obtained through abstractions, which are essentially partitions of the robot configuration space that capture the ability of the robot to steer among the regions in the partition (Belta et al., 2005, 2017). As a result, they suffer from the same scalability issues as the cell-based decomposition methods.

In this article, we address the problem of generating a path for a robot required to satisfy a (global) LTL specification over some known, static service requests, while at the same time servicing a set of locally sensed requests ordered according to their priorities. Consider, for example, a disaster relief scenario requiring an unmanned aircraft to provide persistent surveillance of some affected regions in order to assess the danger posed by unsafe structures with known locations (e.g., by repeatedly taking photos of such regions and uploading the photos at a base region). During flight, by using an onboard camera, the robot looks for survivors and fires. If detected, such requests need to be serviced (e.g., fires need to be extinguished and rescue teams need to be alerted if survivors are detected), possibly with predefined priorities, while making sure that the global, surveillance mission is not compromised. To address the scalability issues mentioned previously, we propose a randomized sampling approach that consists of two components: (1) an off-line algorithm that generates a finite transition system that contains a run satisfying the global specification; and (2) an on-line algorithm that finds local paths that satisfy the local specification, while at the same time making sure that progress is made towards satisfying the global specification.

For the off-line component of the framework, we propose a sampling-based path planning algorithm that finds an infinite path satisfying a LTL formula over a set of properties that hold at some regions in the workspace. The procedure is based on the incremental construction of a transition system in the configuration space followed by the search for one of its satisfying paths. One important feature of the algorithm is that, at a given iteration, it only scales with the number of samples and transitions added to the transitions system at that iteration. This, together with a notion of "sparsity" that we define and enforce on the transition system, play an important role in keeping the overall complexity at a manageable level. In fact, we show that, under some mild assumptions, our definition of sparsity leads to the best possible complexity bound for finding a satisfying path. Finally, while the number of samples increases, the probability that a satisfying path is found approaches 1, i.e., our algorithm is probabilistically complete.

The closest to our proposed off-line algorithm is the work by Karaman and Frazzoli (2009, 2012), where the specifications are given in deterministic μ-calculus. As in this article, Karaman and Frazzoli (2009) guarantee probabilistic completeness and scalability with added samples only at each iteration of their algorithm. However, deterministic μ-calculus formulae have unnatural syntax based on fixed point operators, and are difficult to use by untrained human operators. In contrast, LTL has friendly syntax and semantics, which can be easily translated to natural language (Raman et al., 2013). Note that there is no known procedure to transform a LTL formula ϕ into a μ-calculus formula Ψ such that the size of Ψ is polynomial in the size of ϕ (for details see Cranen et al., 2010). Karaman and Frazzoli (2009) employed the fixed point (Knaster–Tarski) theorem to find a satisfying path. Their method is based on maintaining a "product" graph between the transition system and every sub-formula of their deterministic μ-calculus specification and checking for reachability and the existence of a "type" of cycle on the graph. On the other hand, our algorithm maintains the product automaton between the transition system and a Büchi automaton corresponding to the given LTL specification. Note that, as opposed to LTL model checking (Baier and Katoen, 2008), we use a modified version of product automaton that ensures reachability of the final states. Moreover, we impose that the states of the transition system be bounded away from each other (by a given function decaying in terms of the size of the transition system). Computing sparse structures is also explored by Dobson and Bekris (2013) for PRM using different techniques.

The on-line component of our framework uses sampling-based methods as well. However, in this case the focus is on servicing local request and avoiding local obstacles within the bounded sensing area of the robot, while ensuring the satisfaction of the global specification in the long term. The proposed on-line algorithm is based on the definition of a potential function over the global transition

system that ensures progress toward satisfaction of the global specification. This idea is inspired by Ding et al. (2014). The new algorithm that we propose for the computation of the potential function improves the complexity of the algorithm from Ding et al. (2014) by a polynomial factor. The new algorithm is shown to be correct and to have the same complexity as Dijkstra's algorithm.

The main contribution of this work is a sampling-based, formal framework that combines infinite-time satisfaction of temporal logic global specifications with reactivity to requests sensed locally. Related works include those of Lahijanian et al. (2016), Livingston and Murray (2013), Livingston et al. (2013), Tumova et al. (2013), Ulusoy et al. (2013a), and Vasile et al. (2017a). Lahijanian et al. (2016) considered global specifications given in the more restrictive scLTL fragment of LTL. To deal with the state-space explosion problem, they propose a layered path planning approach that uses a cell decomposition of the configuration space for high-level temporal planning and expansive space trees (EST) for kino-dynamic planning of the low-level, cell-to-cell motion. The on-line algorithm from Tumova et al. (2013) and Reyes Castro et al. (2013) finds minimum violating paths for a robot when the global specification cannot be enforced completely. In Livingston and Murray (2013) and Livingston et al. (2013), the global specifications are given in the GR(1) fragment of LTL, and on-line local re-planning is done through patching invalidated paths based on μ-calculus specifications. Timed specifications such as time window temporal logic (Penedo Álvarez et al., 2016) and signal temporal logic (Vasile et al., 2017a) formulae were considered in conjunction with RRT*, but without considering sensing and reactivity to locally sensed requests. Finally, the idea of using a potential function to enforce the satisfaction of an infinite-time specification through local decisions is inspired from Ding et al. (2014) and Ulusoy et al. (2013a).

This article unifies and extends our previous results published in conference proceedings (Vasile and Belta, 2013, 2014). It includes the entire reactive temporal logic path planning framework. In this article, we expand the proof for the complexity of the off-line algorithm and establish the order of the maximum number of neighbors for a state in the global transition system, which we only hinted to in Vasile and Belta (2013). We also add a conditional result showing that the proposed incremental checking algorithm for satisfying paths is the best possible. This result depends on a conjecture in incremental computation, which is known to be true for a large class of algorithms (Haeupler et al., 2012). Finally, we present new case studies and include experimental trials to validate the framework.

2. Preliminaries

In this section, we briefly review the main concepts from automata theory and formal verification employed in this article. For a detailed exposition of these topics see Baier and Katoen (2008) and the references therein. For a finite set Σ, we use $|\Sigma|$ and 2^{Σ} to denote its cardinality and power set, respectively. Here $\emptyset$ denotes the empty set.

Definition 1. (Deterministic transition system (DTS)). *A weighted deterministic transition system (DTS) is a tuple* $\mathcal{T} = (X, x_0, \Delta, \omega, \Pi, h)$*, where*:

- X *is a finite set of states*;
- $x_0 \in X$ *is the initial state*;
- $\Delta \subseteq X \times X$ *is a set of transitions*;
- $\omega : \Delta \to \mathbb{R}^+$ *is a positive weight function*;
- Π *is a set of properties (atomic propositions)*;
- $h : X \to 2^{\Pi}$ *is a labeling function*.

We also denote a transition $(x, x') \in \Delta$ by $x \to_{\mathcal{T}} x'$. A *trajectory* (or run) of the system is an infinite sequence of states $\mathbf{x} = x_0 x_1 \ldots$ such that $x_k \to_{\mathcal{T}} x_{k+1}$ for all $k \geqslant 0$. A state trajectory $\mathbf{x}$ generates an *output trajectory* $\mathbf{o} = o_0 o_1 \ldots$, where $o_k = h(x_k)$ for all $k \geqslant 0$. The absence of inputs (control actions) in a DTS implicitly means that a transition $(x, x') \in \Delta$ can be chosen deterministically at every state x.

A LTL formula over a set of properties Π is defined using standard Boolean operators, $\neg$ (negation), $\wedge$ (conjunction), and $\vee$ (disjunction), and temporal operators, $\bigcirc$ (next), $\mathcal{U}$ (until), $\diamondsuit$ (eventually), and $\square$ (always). The semantics of LTL formulae over Π are given with respect to infinite words over 2^{Π}, such as the output trajectories of the DTS defined above. Any infinite word satisfying a LTL formula can be written in the form of a finite prefix followed by infinitely many repetitions of a suffix. Verifying whether all output trajectories of a DTS with set of propositions Π satisfy a LTL formula over Π is called LTL model checking. LTL formulae can be used to describe rich mission specifications. For example, formula $\square(\diamondsuit(R_1 \wedge \diamondsuit R_2) \wedge \neg O_1)$ specifies a persistent surveillance task: "visit regions R_1 and R_2 infinitely many times and always avoid obstacle O_1" (see Figure 1). In this article, we consider a particular fragment of LTL, called LTL_{-X} (Baier and Katoen, 2008), which does not include the $\bigcirc$ (next) operator. Formal definitions for the LTL syntax, semantics, and model checking can be found in (Baier and Katoen, 2008).

Definition 2. (Büchi Automaton). *A (non-deterministic) Büchi automaton is a tuple* $\mathcal{B} = (S_{\mathcal{B}}, S_{\mathcal{B}_0}, \Sigma, \delta, F_{\mathcal{B}})$*, where*:

- $S_{\mathcal{B}}$ *is a finite set of states*;
- $S_{\mathcal{B}_0} \subseteq S_{\mathcal{B}}$ *is the set of initial states*;
- Σ *is the input alphabet*;
- $\delta : S_{\mathcal{B}} \times \Sigma \to 2^{S_{\mathcal{B}}}$ *is the transition function*;
- $F_{\mathcal{B}} \subseteq S_{\mathcal{B}}$ *is the set of accepting states*.

A transition $(s, s') \in \delta(s, \sigma)$ is also denoted by $s \xrightarrow{\sigma}_{\mathcal{B}} s'$. A trajectory of the Büchi automaton $s_0 s_1 \ldots$ is generated by an infinite sequence of symbols $\sigma_0 \sigma_1 \ldots$ if $s_0 \in S_{\mathcal{B}_0}$ and $s_k \xrightarrow{\sigma_k}_{\mathcal{B}} s_{k+1}$ for all $k \geqslant 0$. An infinite input sequence over Σ is said to be accepted by a Büchi automaton $\mathcal{B}$ if it

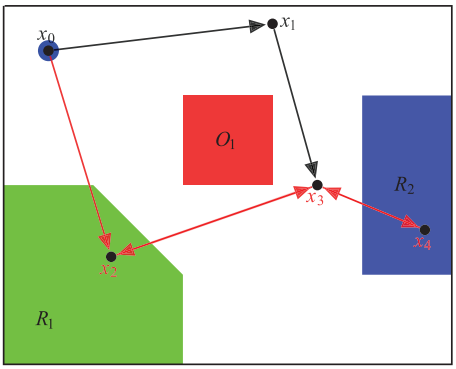


Fig. 1. A simple map with three features: an obstacle O_1 and two regions of interest R_1 and R_2. The mission specification is $\phi = \square(\diamondsuit(R_1 \wedge \diamondsuit R_2) \wedge \neg O_1)$. The initial position of the robot is marked by the blue disk. The graph (in black and red) represents the generated transition system $\mathcal{T}$. The red arrows specify a satisfying trajectory composed of a prefix $[x_0, x_2, x_3]$ and infinitely many repetitions of the suffix $[x_4, x_3, x_2, x_3]$.
Note: Colour version of the figure is available online.

generates at least one trajectory of $\mathcal{B}$ that intersects the set $F_{\mathcal{B}}$ of accepting states infinitely many times.

It was shown in Baier and Katoen (2008) that for every LTL formula ϕ over Π there exists a Büchi automaton $\mathcal{B}$ over alphabet $\Sigma = 2^{\Pi}$ such that $\mathcal{B}$ accepts all and only those infinite sequences over Π that satisfy ϕ. Note that the converse is not true. There exist Büchi automata for which there are no corresponding LTL formulae. However, there are logics such as deterministic μ-calculus which are in one-to-one correspondence with the set of languages accepted by Büchi automata.

The translation problem of LTL formulae to Büchi automata is PSPACE-complete, and the resulting automata sizes can grow exponentially in the size of the logic formulae. However, in practice specifications rarely lead to explosion in automata sizes with respect to the formulae. Moreover, state-of-the-art algorithms such as those in Gastin and Oddoux (2001) and Duret-Lutz et al. (2016) use constructions and heuristic procedures to compute fast small automata. The papers also include analytical and empirical performance analysis.

In this article, we use the translation algorithms off-line, and the resulting automata are independent of the robot models. This means that we can reuse the automata with other robots, scenarios, and synthesis algorithms as long as the specifications remain the same.

Model checking a DTS against a LTL formula is based on the construction of the product automaton between the DTS and the Büchi automaton corresponding to the formula. In this article, we used a modified definition of the product automaton that is optimized for incremental search of a satisfying run. Specifically, the product automaton is defined such that all its states are reachable from the set of initial states.

Definition 3. (Product automaton). *Given a DTS* $\mathcal{T} = (X, x_0, \Delta, \omega, \Pi, h)$ *and a Büchi automaton* $\mathcal{B} = (S_{\mathcal{B}}, S_{\mathcal{B}_0}, 2^{\Pi}, \delta_{\mathcal{B}}, F_{\mathcal{B}})$, *their product automaton, denoted by* $\mathcal{P} = \mathcal{T} \times \mathcal{B}$, *is a tuple* $\mathcal{P} = (S_{\mathcal{P}}, S_{\mathcal{P}_0}, \Delta_{\mathcal{P}}, \omega_{\mathcal{P}}, F_{\mathcal{P}})$ *where*:

- $S_{\mathcal{P}_0} = \{x_0\} \times S_{\mathcal{B}_0}$ *is the set of initial states*;
- $S_{\mathcal{P}} \subseteq X \times S_{\mathcal{B}}$ *is a finite set of states that are reachable from some initial state; for every* $(x^*, s^*) \in S_{\mathcal{P}}$ *there exists a sequence of* $\mathbf{x} = x_0 x_1 \ldots x_n x^*$, *with* $x_k \rightarrow_{\mathcal{T}} x_{k+1}$ for all $0 \leqslant k < n$ and $x_n \rightarrow_{\mathcal{T}} x^*$, *and a sequence* $\mathbf{s} = s_0 s_1 \ldots s_n s^*$ *such that* $s_0 \in S_{\mathcal{B}_0}$, $s_k \overset{h(x_k)}{\rightarrow}_{\mathcal{B}} s_{k+1}$ *for all* $0 \leqslant k < n$ and $s_n \overset{h(x_n)}{\rightarrow}_{\mathcal{T}} s^*$;
- $\Delta_{\mathcal{P}} \subseteq S_{\mathcal{P}} \times S_{\mathcal{P}}$ *is the set of transitions, defined by* $((x,s),(x',s')) \in \Delta_{\mathcal{P}}$ if and only if $x \rightarrow_{\mathcal{T}} x'$ and $s \overset{h(x)}{\rightarrow}_{\mathcal{B}} s'$;
- $\omega_{\mathcal{P}} : \Delta_{\mathcal{P}} \rightarrow \mathbb{R}^+$ *is inherited from* $\mathcal{T}$ *such that* $\omega_{\mathcal{P}}(((x,s),(x',s'))) = \omega((x,x'))$;
- $F_{\mathcal{P}} = (X \times F_{\mathcal{B}}) \cap S_{\mathcal{P}}$ *is the set of accepting states of* $\mathcal{P}$.

A transition in $\mathcal{P}$ is also denoted by $(x,s) \rightarrow_{\mathcal{P}} (x',s')$ if $((x,s),(x',s')) \in \Delta_{\mathcal{P}}$. A trajectory $\mathbf{p} = (x_0,s_0)(x_1,s_1)\ldots$ of $\mathcal{P}$ is an infinite sequence, where $(x_0,s_0) \in S_{\mathcal{P}_0}$ and $(x_k,s_k) \rightarrow_{\mathcal{P}} (x_{k+1},s_{k+1})$ for all $k \geqslant 0$. A trajectory of $\mathcal{P} = \mathcal{T} \times \mathcal{B}$ is said to be accepting if and only if it intersects the set of final states $F_{\mathcal{P}}$ infinitely many times. It follows by construction that a trajectory $\mathbf{p} = (x_0,s_0)(x_1,s_1)\ldots$ of $\mathcal{P}$ is accepting if and only if the trajectory $s_0 s_1 \ldots$ is accepting in $\mathcal{B}$. As a result, a trajectory of $\mathcal{T}$ obtained from an accepting trajectory of $\mathcal{P}$ satisfies the given specification encoded by $\mathcal{B}$. For $x \in X$, we define $\beta_{\mathcal{P}}(x) = \{s \in S_{\mathcal{B}} : (x,s) \in S_{\mathcal{P}}\}$ as the set of Büchi automaton states that correspond to x in $\mathcal{P}$. In addition, we denote the projection of a trajectory $\mathbf{p} = (x_0,s_0)(x_1,s_1)\ldots$ onto $\mathcal{T}$ by $\gamma_{\mathcal{T}}(\mathbf{p}) = x_0 x_1 \ldots$. A similar notation is used for projections of finite trajectories.

For both DTS and automata, we use $|\cdot|$ to denote size, which is the cardinality of the corresponding set of states. A state of a DTS or an automaton is called non-blocking if it has at least one outgoing transition.

3. Problem formulation

Consider a robot moving in an environment (workspace) $\mathcal{D}$ containing a set of disjoint regions of interest $\mathcal{R}_G$. We assume that the robot can precisely localize itself in the environment. There is a set of service requests Π_G at the regions in $\mathcal{R}_G$ and their location is given by a map $\mathcal{L}_G : \mathcal{R}_G \rightarrow 2^{\Pi_G}$. We assume that these regions as well as the labeling map are static and a priori known to the robot. We refer to these as *global* regions and requests, because these are used to define the long-term goal of the robot's mission. An example of an environment with global regions and requests is shown in Figure 2.

While the robot moves in the environment, it can locally sense a set of dynamic service requests denoted by Π_L and a particular type of avoidance request denoted by π_O, which captures moving obstacles, unsafe areas, etc. We assume $\Pi_G \cap (\Pi_L \cup \{\pi_O\}) = \emptyset$. A dynamic request from Π_L occurs at a point in the environment and has an associated

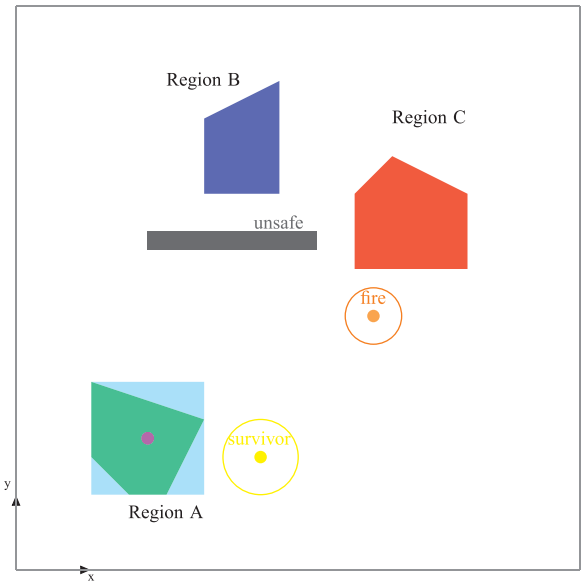


Fig. 2. Simplified representation of a disaster scenario considered in Example 5. The environment contains three global regions A, B and C colored in green, blue, and red, respectively. Three dynamic requests are also shown as colored points: a *survivor* (yellow), a *fire* (orange), and a local obstacle (black). The circles around them delimit the corresponding servicing areas. The initial position of the robot is shown in magenta and the cyan rectangle corresponds to its sensing area. In this figure the robot does not detect any dynamic request or local obstacles. Note: Colour version of the figure is available online.

servicing radius, which specifies the maximum distance from which the robot can service it. The servicing radius of a request is determined by its type (Π_L) and all servicing radii are known a priori. The robot may service a dynamic request by moving inside the request's servicing radius and performing an appropriate action. Thus, a dynamic request is define by its time-varying position in the environment and its type from Π_L. Multiple dynamic requests of the same type may be present in the environment at the same time. The number, type, and motion of dynamic requests and local obstacles are *a priori* unknown. Moreover, it is unknown when and where dynamic requests appear in the environment. We assume that once a request is serviced, it disappears from the environment.

The region around the robot in which the robot can sense a dynamic request, including π_O, is called the *sensing area* of the corresponding sensor. For simplicity, we assume that all sensors have the same sensing area. The sensing area may be of any shape and size provided that it is connected and full-dimensional (see Figure 2). We assume that the avoidance request π_O is associated with whole regions, parts of which can be detected when they intersect with the robot's sensing area. For simplicity, we refer to regions satisfying π_O as *local obstacles*. We do not assume that sensors have the ability to distinguish between multiple requests of the same type (aside from location), i.e., assign identities to dynamic requests and local obstacles. This means that the robot at the current time only knows the requests within its sensing area, and for each request it knows its position and type; for local obstacles it knows only their extend within the sensing area. This also implies that the robot cannot establish whether a request entering and leaving the sensing area multiple times is the same request. Tracking is outside the scope of this article, but has been considered in Serlin et al. (2018a,b) with LTL constraints, and in Pierson et al. (2016) as active pursuit with exclusion zones. The set of regions corresponding to local obstacles present in the environment at time $t \geqslant 0$ is denoted by $\mathcal{R}_L(t)$.

The mission specification is composed of two parts: a *global mission specification*, which is defined over the set of global properties Π_G, and a *local mission specification*, which specifies how on-line detected requests Π_L must be handled. The global mission specification, which defines the long-term motion of the robot, is given as a LTL$_{-X}$ formula Φ_G. When the robot passes over a global region, it is assumed that the robot services the requests associated with the region. Formally, a symbol in 2^{Π_G} is generated whenever the robot is inside the boundary of a region in $\mathcal{R}_G$, and $\emptyset$ is generated when the robot is on the exterior of all regions. In Figure 1, the symbol generated at x_0, x_1, and x_3 is $\mathcal{L}_G(x_0) = \mathcal{L}_G(x_1) = \mathcal{L}_G(x_3) = \emptyset$, while those for x_2 and x_4 are $\mathcal{L}_G(x_2) = \{R_1\}$ and $\mathcal{L}_G(x_4) = \{R_2\}$, respectively. Therefore, a path traveled by the robot generates a word over Π_G. A path is said to *satisfy the global mission specification* Φ_G if the corresponding word satisfies Φ_G. The local mission specification is "Service the highest priority, locally sensed dynamic request, while avoiding local obstacles" evaluated at the current time. Priorities are given by an injective function $prio : \Pi_L \to \mathbb{N}$ that assigns lower values to higher-priority requests. If the robot detects dynamic requests, it must go and service the request with the highest priority. If multiple requests have the same (highest) priority, then the robot can choose any of them. As the local specification is evaluated at the current time of the robot, the highest-priority request may change before the servicing of a previously considered request. In addition, the robot must avoid all local obstacles marked by π_O.

Planning is performed in the configuration space of the robot. Let $\mathcal{C}$ be the compact configuration space of the robot and $\mathcal{H} : \mathcal{C} \to \mathcal{D}$ be a submersion that maps each configuration x to a position $y = \mathcal{H}(x) \in \mathcal{D}$. Formally, the problem can be formulated as follows.

Problem 4. *Given a partially known environment described by* $(\mathcal{D}, \mathcal{R}_G, \Pi_G, \mathcal{L}_G, \Pi_L)$*, an initial configuration* $x_0 \in \mathcal{C}$*, a LTL*$_{-X}$ *formula* Φ_G *over the set of properties* Π_G*, and a priority function* $prio : \Pi_L \to \mathbb{N}$*, find an (infinite) path in the configuration space* $\mathcal{C}$ *originating at* x_0 *such that the path* $\mathbf{y} = \mathcal{H}(\mathbf{x})$ *in the environment satisfies* Φ_G *and on-line detected dynamic requests, while avoiding local obstacles.*

Note that in this article we are concerned with computing satisfying paths in the robots' configuration spaces and focus on correctness with respect to global and local specifications. Path tracking algorithms can be used to implement the

computed paths on robots, and take into account their dynamics (Aguiar and Hespanha, 2007; Frazzoli et al., 2000; Kuwata et al., 2009; Murray et al., 1994).

Example 5. *Figure 2 shows a simplified disaster response scenario, in which a fully actuated point robot is deployed in an environment where three global regions of interest A, B, and C are defined. The set of dynamic requests is* $\Pi_L = \{fire, survivor\}$ *and the local obstacle is* $\pi_O = unsafe$. *If the robot detects requests fire or survivor, it must service them by going within the corresponding servicing radii and initiating appropriate actions (i.e., extinguishing the fire and providing medical relief, respectively). If the robot detects the local obstacle unsafe (shown in black in Figure 2), the robot must avoid that region. The limited sensing area of the robot's sensors is depicted in Figure 2 by a cyan rectangle.*

The global mission specification is: "*Go to region A and then go to regions B or C infinitely often.*" *This specification can be expressed in LTL*$_{-X}$ *as*

$$\Phi_G := \square(\diamondsuit(A \wedge \diamondsuit(B \vee C))) \tag{1}$$

The local mission specification is to "Extinguish fires and provide medical assistance to survivors, with priority given to survivors, while avoiding unsafe areas." Thus, the priority function is defined such that $prio(survivor) = 0$ and $prio(fire) = 1$.

4. Outline of the approach

We propose a computational framework to solve Problem 4 that consists of two parts: (a) an *off-line* sampling-based algorithm to compute a global transition system $\mathcal{T}_G$ in the configuration space $\mathcal{C}$ of the robot that contains a path whose image in the workspace $\mathcal{D}$ satisfies the global mission specification Φ_G; and (b) an *on-line* sampling-based algorithm that computes at every time step a local control strategy that takes into account dynamic requests such that both local and global mission specifications are met.

A possible approach to the off-line part of Problem 4 is to construct a partition of the configuration space such that its image in the workspace contains the regions of interest as elements of the partition. By using input–output linearizations and vector field assignments in the regions of the partition, it was shown that "equivalent" abstractions in the form of finite (not necessarily deterministic) transition systems can be constructed for a large variety of robot dynamics that include car-like vehicles and quadrotors (Belta et al., 2005; Lindemann and LaValle, 2009; Ulusoy et al., 2013b). Model checking and automata game techniques can then be used to control the abstractions from the temporal logic specification (Kloetzer and Belta, 2008). The main limitation of this approach is its high complexity, as both the synthesis and abstraction algorithms scale at least exponentially with the dimension of the configuration space.

In this article, we propose a sampling-based algorithm for the construction of $\mathcal{T}_G$ that can be summarized as follows: (1) the LTL formula ϕ_G is translated to a Büchi automaton $\mathcal{B}$; (2) a transition system $\mathcal{T}_G$ is incrementally constructed from the initial configuration x_0 using an RRG-based algorithm; (3) concurrently with (2), the product automaton $\mathcal{P}_G = \mathcal{T}_G \times \mathcal{B}$ is updated and used to check whether there is a trajectory of $\mathcal{T}_G$ that satisfies Φ_G. As will become clear later, our proposed algorithm is *probabilistically complete* (Karaman and Frazzoli, 2011; LaValle, 2006) (i.e., it finds a solution with probability 1 if one exists and the number of samples approaches infinity) and the resulting transition system $\mathcal{T}_G$ is *sparse* (i.e., its states are "far" away from each other). In addition, it is incremental, in the sense that its complexity scales only with the number of samples generated at the current iteration, rather than with size of $\mathcal{T}_G$.

The proposed approach to the on-line part of Problem 4 is based on the RRT algorithm, a probabilistically complete sampling-based path planning method. RRT randomly grows trees instead of general graphs. We modify the standard RRT in order to find local paths which preserve the satisfaction of the global specification Φ_G, while servicing on-line requests and avoiding locally sensed obstacles. We use ideas from Bauer et al. (2007) on monitors for LTL formulae and Ding et al. (2014) on potential functions to ensure the correctness of the local random paths with respect to Φ_G.

5. Solution

In the following, we denote by $\mathcal{T}_G = (X_G, x_0, \Delta_G, \omega_G, \Pi_G, h_G)$ the global transition system, by $\mathcal{B}$ the Büchi automaton encoding the LTL$_{-X}$ formula Φ_G and by $\mathcal{P}_G = \mathcal{T}_G \times \mathcal{B}$ their product. The local transition system is given by $\mathcal{T}_L = (X_L, x_c, \Delta_L, \omega_L, \Pi_L \cup \{\pi_O\}, h_L)$ which is incrementally generated at each time step of the on-line procedure (see Section 5.2) from the current configuration x_c. An element of $\mathcal{D}$ will be called a *position*. The states of $\mathcal{T}_G$ and $\mathcal{T}_L$ are configurations in $\mathcal{C}$. The weight of a transition of $\mathcal{T}_G$ or $\mathcal{T}_L$ is given by the distance between its endpoints in $\mathcal{C}$. The labeling function $h_G(x)$, $x \in X_G$, is defined as the proposition set corresponding to the region the projection of x belong to. Formally, $h_G(x) = \mathcal{L}_G(R)$ if $\mathcal{H}(x) \in R$ for some $R \in \mathcal{R}_G$, and $h_G(x) = \emptyset$ otherwise. Similarly, the labeling function $h_L(x)$, $x \in X_L$ is defined as the set of local requests which are satisfied at position $y = \mathcal{H}(x)$ if $y \notin \mathcal{R}_L(t)$, and $h_L(x) = \pi_O$, otherwise. Recall that the robot has knowledge only about the local requests and obstacles inside its sensing area, which is determined by the current position $\mathcal{H}(x_c)$. In addition, $h_L(x)$ may be $\emptyset$ if no local requests are satisfied by the corresponding position $y = \mathcal{H}(x)$ and y does not fall inside a local obstacle.

We make the following additional assumptions that are necessary in the technical treatment presented in the following. For a set $R \subseteq \mathcal{D}$ that is connected and has full dimension in $\mathcal{D}$, we assume that the inverse set $\mathcal{H}^{-1}(R)$ also has

full dimension in $\mathcal{C}$. The global regions and local obstacles are connected sets with non-empty interior (i.e., they have full dimension in $\mathcal{D}$). In addition, all the connected regions in the free space, between global regions and obstacles, respectively, are full dimensional. This implies that all global regions, local obstacles, service areas for dynamic requests, and connected free-space regions (all subsets of $\mathcal{D}$) have corresponding inverse sets (through $\mathcal{H}^{-1}$) of non-zero Lebesgue measure in $\mathcal{C}$. It is important to note that these are just technical assumptions, which are normally made in sampling-based approaches, and we do not need to construct the inverse map $\mathcal{H}^{-1}$. In the sampling-based algorithms described in the following, we only need to check how the environment image of a configuration satisfies features of interest in the environment. Finally, we assume that the robot knows its configuration precisely and it can follow trajectories in the configuration space made of connected line segments. The initial configuration x_0 of the robot is known and $\mathcal{H}(x_0) = y_0$.

5.1. Off-line algorithm

The starting point for our solution to Problem 4 is the off-line algorithm to generate the global transition system $\mathcal{T}_G$. The algorithm is based on the RRG algorithm, which is an extension of RRT (Karaman and Frazzoli, 2011) that maintains a digraph instead of a tree, and can therefore be used as a model for general ω-regular languages (Karaman and Frazzoli, 2009). However, we modify the RRG to obtain a "sparse" transition system that satisfies a given LTL formula Φ_G. More precisely, a transition system $\mathcal{T}_G$ is "sparse" if the minimum distance between any two states of $\mathcal{T}$ is greater than a prescribed function dependent only on the size of $\mathcal{T}_G$ ($\min_{x,x' \in \mathcal{T}_G} \|x - x'\|_2 \geqslant \eta(|\mathcal{T}_G|)$). The distance used to define sparsity is inherited from the underlying configuration space and is not related to the graph theoretical distance between states in $\mathcal{T}_G$. Throughout this article, we assume that this distance is Euclidean.

As stated in Section 4, sparsity of $\mathcal{T}_G$ is desired because the transition system is then used in the on-line part of the framework. The environment is partially known by the robot before the start of the mission. As transitions of $\mathcal{T}_G$ may need to be locally re-planned on-line, $\mathcal{T}_G$ must only capture the essential features of $\mathcal{D}$ such that Φ_G is satisfied. Sparseness also plays an important role in establishing the complexity bounds for the incremental search algorithm (see Section 5.1.3).

5.1.1. Primitive functions. We first briefly introduce the functions used by the algorithm.

Sampling function The algorithm has access to a sampling function $sample : \mathbb{N} \to \mathcal{C}$, which generates independent and identically distributed samples from a given distribution P. We assume that the support of P is the entire configuration space $\mathcal{C}$.

Steer function The steer function $steer : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined based on the robot's dynamics.[1] Given a configuration x and goal configuration x_g, it returns a new configuration x_n that can be reached from x by following the dynamics of the robot and that satisfies $\|x_n - x_g\|_2 < \|x - x_g\|_2$. If a third parameter η_L is given, then x_n must be within η_L distance away from x, $\|x_n - x\|_2 < \eta_L$.

Near function The function $near : \mathcal{C} \times \mathbb{R} \to 2^X$ is a function of a configuration x and a parameter η_2, which returns the set of states from the transition system $\mathcal{T}_G$ that are at most at η_2 distance away from x. In other words, *near* returns all states in $\mathcal{T}_G$ that are inside the n-dimensional sphere of center x and radius η_2.

Far function The function $far : \mathcal{C} \times \mathbb{R} \times \mathbb{R} \to 2^X$ is a function of a configuration x and two parameters η_1 and η_2. It returns the set of states from the transition system $\mathcal{T}_G$ that are at most at η_2 distance away from x. However, the difference from the *near* function is that *far* returns an empty set if any state of $\mathcal{T}_G$ is closer to x than η_1. Geometrically, this means that *far* returns a non-empty set for a given state x if there are states in $\mathcal{T}_G$, which are inside the n-dimensional sphere of center x and radius η_2 and all states of $\mathcal{T}_G$ are outside the sphere with the same center, but radius η_1. Thus, x has to be "far" away from all states in its immediate neighborhood (see Figure 3). This function is used to achieve the "sparseness" of the resulting transition system.

isSimpleSegment function The function $isSimpleSegment : \mathcal{C} \times \mathcal{C} \to \{0, 1\}$ is a function that takes two configurations x_1, x_2 in $\mathcal{C}$ and returns 1 if the line segment $[x_1, x_2]$ ($\{x \in \mathbb{R}^n : x = \lambda x_1 + (1 - \lambda)x_2, \lambda \in [0, 1]\}$) is simple, otherwise it returns 0. Let $y_1 = \mathcal{H}(x_1)$, $y_2 = \mathcal{H}(x_2)$ and $[y_1, y_2] = \mathcal{H}([x_1, x_2])$ be the projections of x_1, x_2 and the line segment $[x_1, x_2]$ onto the workspace $\mathcal{D}$, respectively. A line segment $[x_1, x_2]$ is simple if $[x_1, x_2] \subset \mathcal{C}$ and the number of times $[y_1, y_2]$ crosses the boundary of any region $R \in \mathcal{R}$ is at most one. Therefore, *isSimpleSegment* returns 1 if either: (1) y_1 and y_2 belong to the same region R and $[y_1, y_2]$ does not cross the boundary of R or (2) y_1 and y_2 belong to two regions R_1 and R_2, respectively, and $[y_1, y_2]$ crosses the common boundary of R_1 and R_2 once. Here R or at most one of R_1 and R_2 may be a free space region (a connected set in $\mathcal{D} \setminus \bigcup_{R \in \mathcal{R}} R$). See Figure 3 for examples. In Algorithm 1, a transition is rejected if it corresponds to a non-simple line segment (i.e., *isSimpleSegment* function returns 0).

Bound functions The functions $\eta_1 : \mathbb{Z}+ \to \mathbb{R}$ (lower bound) and $\eta_2 : \mathbb{Z}+ \to \mathbb{R}$ (upper bound) define the bounds on the distance between a configuration in $\mathcal{C}$ and the states of the transition system $\mathcal{T}_G$ in terms of the size of $\mathcal{T}_G$. These are used as parameters for functions *far* and *near*. We impose $\eta_1(k) < \eta_2(k)$ for all $k \geqslant 1$. We also assume that $c\,\eta_1(k) > \eta_2(k)$, for some finite $c > 1$ and all $k \geqslant 0$. In addition, η_1 tends to zero as k tends to infinity. The rate of decay of $\eta_1(\cdot)$ has to be fast enough such that a new sample may be generated. Specifically, the set of all configurations where the center of an n-sphere of radius $\eta_1/2$ may be placed such that it does not intersect any of

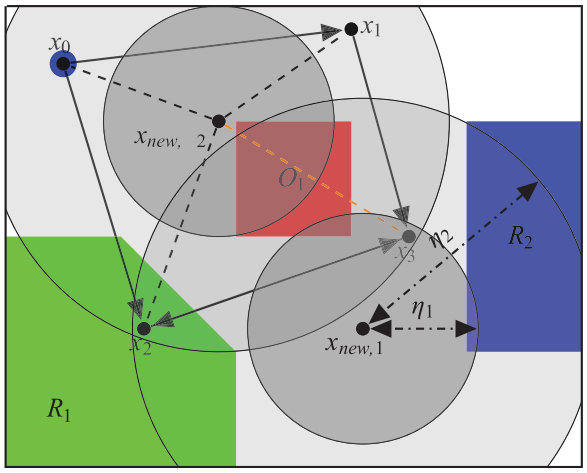


Fig. 3. A simple map with three features: an obstacle O_1 and two regions R_1, R_2. The robot is assumed to be a fully actuated point and $\mathcal{C} = \mathcal{D} \subset \mathbb{R}^2$. At the current iteration the states of $\mathcal{T}_G$ are $\{x_0, x_1, x_2, x_3\}$. The transitions of $\mathcal{T}_G$ are represented by the black arrows. The initial configuration is x_0 and is marked by the blue disk. The radii of the dark gray (inner) disks and the light gray (outer) disks are η_1 and η_2, respectively. A new sample $x_{new,1} \in \mathcal{C}$ is generated, but it will not be considered as a potential new state of $\mathcal{T}_G$, because it is within η_1 distance from state 3 ($far(x_{new,1}, \eta_1, \eta_2) = \emptyset$). Another sample $x_{new,2} \in \mathcal{C}$ is generated, which is at least η_1 distance away from all states in $\mathcal{T}_G$. In this case, $far(x_{new,2}, \eta_1, \eta_2) = \{x_0, x_1, x_2, x_3\}$ and the algorithm attempts to create transition to and from the new sample $x_{new,2}$. The transitions $\{(x_{new,2}, x_0), (x_0, x_{new,2}), (x_{new,2}, x_1), (x_1, x_{new,2}), (x_{new,2}, x_2), (x_2, x_{new,2})\}$ (marked by black dashed lines) are added to $\mathcal{T}_G$, because all these transitions correspond to simple line segments (*isSimpleSegment* returns 1 for all of them). For example, $isSimpleSegment(x_{new,2}, x_0) = 1$, because $x_{new,2}$ and x_0 belong to the same region (the free-space region) and $[x_{new,2}, x_0]$ does not intersect any other region. Here $isSimpleSegment(x_{new,2}, x_2) = 1$, because $[x_{new,2}, x_2]$ crosses the boundary between the free-space region and region R_1 once. On the other hand, the transitions $\{(x_{new,2}, x_3), (x_3, x_{new,2})\}$ (marked by orange dashed lines) are not added to $\mathcal{T}_G$, since they pass over the obstacle O_1. In this case, $isSimpleSegment(x_3, x_{new,2}) = 0$, because x_3 and $x_{new,2}$ are in the same region, but $[x_3, x_{new,2}]$ crosses the boundary of O_1 twice. Note: Colour version of the figure is available online.

the d-spheres corresponding to the states in $\mathcal{T}_G$ has to have non-zero measure with respect to the probability measure P used by the sampling function. One conservative upper bound is $\eta_1(k) < \frac{1}{\sqrt{\pi}} \sqrt[d]{\frac{\mu(\mathcal{C})\Gamma(d/2+1)}{k}}$ for all $k \geqslant 1$, where $\mu(\mathcal{C})$ is the total measure (volume) of the configuration space, d is the dimension of $\mathcal{C}$, and Γ is the gamma function. This bound corresponds to the case when $\mathcal{C}$ is convex and there is enough space to insert an n-sphere of radius $\eta_1/2$ between every two distinct states of $\mathcal{T}_G$. To simplify the notation, we drop the parameter for these functions and assume that k is always given by the current size of the transition system, $k = |\mathcal{T}|$.

5.1.2. Sparse RRG. The goal of the modified RRG algorithm (see Algorithm 1) is to find a satisfying run, but such that the resulting transition system is "sparse," i.e., states are "sufficiently" apart from each other. The algorithm iterates until a satisfying run originating in x_0 is found.

At each iteration, a new sample x_r is generated (line 5 in Algorithm 1). For each state x in $\mathcal{T}_G$ that is "far" from the sample x_r ($x \in far(x_r, \eta_1, \eta_2)$), a new configuration x'_r is computed such that the robot can be steered from x to x'_r and the distance to x_r is decreased (line 8). The two loops of the algorithm (lines 7–13 and 16–21) are executed if and only if the *far* function returns a non-empty set. However, x'_r is regarded as a potential new state of $\mathcal{T}_G$, and not x_r. Thus, the *steer* function plays an important role in the "sparsity" of the final transition system. Next, it is checked if the potential new transition (x, x'_r) is a simple segment (line 9). It is also verified whether x'_r may lead to a solution, which is equivalent to testing whether x'_r induces at least one non-blocking state in $\mathcal{P}_G$ (see Algorithm 2). If configuration x'_r and the corresponding transition (x, x'_r) pass all tests, then they are added to the list of new states and list of new transitions of $\mathcal{T}_G$, respectively (lines 12–13).

After all "far" neighbors of x_r are processed, the transition system is updated. Note that at this point $\mathcal{T}_G$ was only extended with states that explore "new space." However, in order to model ω-regular languages the algorithm must also close cycles. Therefore, the same procedure as before (lines 6–14) is also applied to the newly added states $X_{G'}$ (lines 15–22 of Algorithm 1). The difference is that it is checked whether states from $X_{G'}$ can steer the robot back to states in $\mathcal{T}_G$ in order to close cycles. In addition, because we know that the states in $X_{G'}$ are "far" from their neighbors, the *near* function will be used instead of the *far* function. The algorithm returns a (prefix, suffix) pair in $\mathcal{T}_G$ obtained by projection from the corresponding path $(p_0 \xrightarrow{*}_{\mathcal{P}_G} p_F)$ and cycle $(p_F \xrightarrow{+}_{\mathcal{P}_G} p_F)$ in $\mathcal{P}_G$, respectively. The $*$ above the transition symbol means that the length of the path can be 0 or more, while $+$ denotes that the length of the cycle must be at least 1.

In the end, the result is a transition system $\mathcal{T}_G$ that captures the general topology of the environment. In the next section, we show that $\mathcal{T}_G$ also yields a run that satisfies the given specification.

5.1.3. Incremental search for a satisfying run. The proposed approach of incrementally constructing a transition system raises the problem of how to efficiently check for a satisfying run at each iteration. As mentioned in the previous section, the search for satisfying runs is performed on the product automaton. Note that testing whether there exists a trajectory of $\mathcal{T}_G$ from the initial configuration x_0 that satisfies the given LTL$_{-X}$ formula Φ_G is equivalent to searching for a path from an initial state p_0 to a final state p_F in the product automaton $\mathcal{P}_G = \mathcal{T}_G \times \mathcal{B}$ and for a cycle containing p_F of length greater than 1, where $\mathcal{B}$ is the Büchi automaton corresponding to Φ_G. If such a path and a cycle are found, then their projection onto $\mathcal{T}_G$ represents a satisfying infinite trajectory (line 23 of Algorithm 1).

Algorithm 1: Sparse RRG

Input: $\mathcal{B}$– Büchi automaton corresponding to Φ_G
Input: x_0 initial configuration of the robot
Output: (prefix, suffix) in $\mathcal{T}_G$
1 Construct $\mathcal{T}_G$ with x_0 as initial state
2 Construct $\mathcal{P}_G = \mathcal{T}_G \times \mathcal{B}$
3 Initialize $scc(\cdot)$
4 **while** $\neg(x_0 \models \phi)$ ($\equiv\neg(\exists p \in F_{\mathcal{P}_G}$ s.t. $|scc(p)|>1)$) **do**
5 $x_r \leftarrow sample()$
6 $X'_G \leftarrow \emptyset$, $\Delta'_G \leftarrow \emptyset$
7 **foreach** $x \in far(x_r, \eta_1, \eta_2)$ **do**
8 $x'_r \leftarrow steer(x, x_r)$
9 **if** $isSimpleSegment(x_r, x'_r)$ **then**
10 $added \leftarrow updatePA(\mathcal{P}_G, \mathcal{B}, (x, x'_r))$
11 **if** *added is True* **then**
12 $X'_G \leftarrow X'_G \cup \{x'_r\}$
13 $\Delta'_G \leftarrow \Delta'_G \cup \{(x, x'_r)\}$
14 $\mathcal{T}_G \leftarrow \mathcal{T}_G \cup (X'_G, \Delta'_G)$
15 $\Delta'_G \leftarrow \emptyset$
16 **foreach** $x'_r \in X'_G$ **do**
17 **foreach** $x \in near(x'_r, \eta_2)$ **do**
18 **if** $(x = steer(x'_r, x)) \wedge isSimpleSegment(x'_r, x)$ **then**
19 $added \leftarrow updatePA(\mathcal{P}_G, \mathcal{B}, (x, x'_r))$
20 **if** *added is True* **then**
21 $\Delta'_G \leftarrow \Delta'_G \cup \{(x'_r, x)\}$
22 $\mathcal{T}_G \leftarrow \mathcal{T}_G \cup (X'_G, \Delta'_G)$
23 **return** $(\gamma_{\mathcal{T}_G}(p_0 \xrightarrow{*}_{\mathcal{P}_G} p_F), \gamma_{\mathcal{T}_G}(p_F \xrightarrow{+}_{\mathcal{P}_G} p_F))$, *where* $p_F \in F_{\mathcal{P}}$

Testing whether p_F belongs to a non-degenerate cycle (length greater than 1) is equivalent to testing whether p_F belong to a non-trivial strongly connected component: SCC (the size of the SCC is greater than 1). Checking for a satisfying trajectory in $\mathcal{P}_G$ is performed incrementally as the transition system is modified.

The reachability of the final states from initial ones in $\mathcal{P}_G$ is guaranteed by construction (see Definition 3). However, we need to define a procedure (see Algorithm 2) to incrementally update $\mathcal{P}_G$ when a new transition is added to $\mathcal{T}_G$. Consider the (non-incremental) case of constructing $\mathcal{P}_G = \mathcal{T}_G \times \mathcal{B}$. This is done by a traversal of $\bar{\mathcal{P}}_G = (X_G \times S_{\mathcal{B}}, \bar{\Delta}_{\mathcal{P}_G})$ from all initial states, where $((x,s),(x',s')) \in \bar{\Delta}_{\mathcal{P}_G}$ if $x \rightarrow_{\mathcal{T}_G} x'$ and $s \xrightarrow{h_G(x)} s'$. Here $\bar{\mathcal{P}}_G$ is a product automaton but without the reachability requirement. This suggests that the way to update $\mathcal{P}_G$ when a transition (x,x') is added to $\mathcal{T}_G$, is to do a traversal from all states p of $\mathcal{P}_G$ such that $\gamma_{\mathcal{T}_G}(p) = x$. In addition, it is checked whether x' induces any non-blocking states in $\mathcal{P}_G$ (lines 1–3 of Algorithm 2). The test is performed by computing the set $S'_{\mathcal{P}_G}$ of non-blocking states of $\mathcal{P}_G$ (line 1) such that $p' \in S'_{\mathcal{P}_G}$ has $\gamma_{\mathcal{T}_G}(p') = x'$ and p' is obtained by a transition from $\{(x,s) : s \in \beta_{\mathcal{P}_G}(x)\}$. If $S'_{\mathcal{P}_G}$ is empty, then the transition (x,x') of $\mathcal{T}_G$ is discarded and the procedure stops (line 3). Otherwise, the product automaton $\mathcal{P}_G$ is updated recursively to add all states that become reachable because of the states in $S'_{\mathcal{P}_G}$. The recursive procedure is performed from each state in $S'_{\mathcal{P}_G}$ as follows: if a state p (line 9) is not in $\mathcal{P}_G$, then it is added to $\mathcal{P}_G$ together with all its outgoing transitions (line 4) and the recursive procedure continues from the outgoing states of p; if p is in $\mathcal{P}_G$, then the traversal stops, but its outgoing transitions are still added to $\mathcal{P}_G$ (line 14). The incremental construction of $\mathcal{P}_G$ has the same overall complexity as constructing $\mathcal{P}_G$ from the final $\mathcal{T}_G$ and $\mathcal{B}$, because the recursive procedure just performs traversals that do not visit states already in $\mathcal{P}_G$. Thus, we focus our complexity analysis on the next step of the incremental search algorithm.

The second part of the incremental search procedure is concerned with maintaining the SCCs of $\mathcal{P}_G$ (line 16 of Algorithm 2) as new transitions are added (these are stored in $\Delta'_{\mathcal{P}_G}$ in Algorithm 2). To incrementally maintain the SCCs of the product automaton, we employ the soft-threshold-search algorithm presented in Haeupler et al. (2012). The algorithm maintains a topological order of the super-vertices corresponding to each SCC. When a new transition is added to $\mathcal{P}_G$, the algorithm proceeds to re-establish a topological order and merges vertices if new SCCs are formed. The details of the algorithm are presented in Haeupler et al. (2012). The authors also offered insight into the complexity of the algorithm. They showed that, under a mild assumption, the incremental algorithm has the best possible complexity bound.

Incrementally maintaining $\mathcal{P}_G$ and its SCCs yields a quick way to check whether a trajectory of $\mathcal{T}_G$ satisfies Φ_G (line 4 of Algorithm 1). Theorem 7 establishes the overall complexity of Algorithm 2.

5.1.4. Complexity of the off-line algorithm. In this section, the overall complexity of Algorithm 2 is established and we show that this is the best possible under some mild assumptions. The proofs of Theorems 7 and 11 are based on the analysis from Haeupler et al. (2012) of incremental algorithms for cycle detection and maintenance of SCCs.

Remark 6. *The complexity results in this section are reported with respect to the size of transition systems, while the size of the Büchi automata is considered a fixed parameter. The translation step is performed once and is decoupled from the incremental procedure in the sense that the Büchi automata do not change. Details on the LTL-to-Büchi translation problem and algorithms can be found in Gastin and Oddoux (2001) and Duret-Lutz et al. (2016).*

Algorithm 2 uses the soft-threshold-search algorithm presented in Haeupler et al. (2012) to incrementally maintain SCCs. The soft-threshold-search algorithm has $O(m^{\frac{3}{2}})$ complexity and is very efficient for sparse graphs (in asymptotic sense), where m is the number of edges added to $\mathcal{T}_G$. Recall that a graph is sparse if the number of edges m is asymptotically the same as the number of nodes n, i.e., $m = O(n)$.

Theorem 7. *The overall execution time of the incremental search algorithm (Algorithm 2) is $O(n^{\frac{3}{2}})$, where $n = |\mathcal{T}_G|$ is the number of states added to $\mathcal{T}_G$ in Algorithm 1.*

Remark 8. *First, note that the execution time of the incremental procedure is better by a polynomial factor than*

Algorithm 2: Incremental Search for a Satisfying Run

Input: $\mathcal{P}_G$ – product automaton
Input: $\mathcal{B}$ – Büchi automaton
Input: (x, x')– new transition in $\mathcal{T}_G$
Output: Boolean value – indicates if $\mathcal{P}_G$ was modified

$S'_{\mathcal{P}_G} \leftarrow \{(x', s') : s \overset{h_G(x)}{\rightarrow}_{\mathcal{B}} s', s \in \beta_{\mathcal{P}_G}(x), s' \text{ non-blocking}\}$
$\Delta'_{\mathcal{P}_G} \leftarrow \{((x, s), (x', s')) : s \in \beta_{\mathcal{P}_G}(x), s \overset{h_G(x)}{\rightarrow}_{\mathcal{B}} s',\ (x', s') \in S'_{\mathcal{P}_G}\}$
if $S'_{\mathcal{P}_G} \neq \emptyset$ **then**
  $\mathcal{P}_G \leftarrow \mathcal{P}_G \cup (S'_{\mathcal{P}_G}, \Delta'_{\mathcal{P}_G})$
  $stack \leftarrow S'_{\mathcal{P}_G}$
  while $stack \neq \emptyset$ **do**
    $p_1 = (x_1, s_1) \leftarrow stack.pop()$
    foreach $p_2 \in \{(x_2, s_2) : x_1 \rightarrow_{\mathcal{T}_G} x_2, s_1 \overset{h_G(x_1)}{\rightarrow}_{\mathcal{B}} s_2\}$ **do**
      if $p_2 \notin S_{\mathcal{P}_G}$ **then**
        $\mathcal{P}_G \leftarrow \mathcal{P}_G \cup (\{p_2\}, \{(p_1, p_2)\})$
        $\Delta'_{\mathcal{P}_G} \leftarrow \Delta'_{\mathcal{P}_G} \cup \{(p_1, p_2)\}$
        $stack \leftarrow stack \cup \{p_2\}$
      else if $(p_1, p_2) \notin \Delta_{\mathcal{P}_G}$ then
        $\Delta_{\mathcal{P}_G} \leftarrow \Delta_{\mathcal{P}_G} \cup \{(p_1, p_2)\}$
        $\Delta'_{\mathcal{P}_G} \leftarrow \Delta'_{\mathcal{P}_G} \cup \{(p_1, p_2)\}$
  updateSCC($\mathcal{P}$, scc, $\Delta'_{\mathcal{P}_G}$)
  return True
return False

naïvely running a linear-time SCC algorithm at each step, since this will have complexity $O(m^2)$, where $m = |\Delta_G|$. The algorithm presented in Haeupler et al. (2012) improves the previously best known bound by a logarithmic factor (for sparse graphs). The proof of Theorem 7 exploits the fact that the "sparseness" (metric) property we defined implies a topological sparseness, i.e., $\mathcal{T}_G$ is a sparse graph.

Proof. The soft-threshold-search attains the desired complexity only for sparse graphs. Therefore, what we need to show is that the transition system generated by Algorithm 1 is a sparse graph. Note that although we run the SCC algorithm on the product automaton, the asymptotic execution time is not affected by analyzing the transition system instead of the product automaton, because the Büchi automaton is fixed. This follows from $|S_{\mathcal{P}_G}| \leqslant |S_{\mathcal{B}}| \cdot |X_G|$ and $|\Delta_{\mathcal{P}_G}| \leqslant |\delta_{\mathcal{B}}| \cdot |\Delta_G|$.

Intuitively, the underlying graph of $\mathcal{T}_G$ is sparse, because the states were generated "far" from each other. When a new state is added to $\mathcal{T}_G$, it will be connected to other states that are at least η_1 and at most η_2 distance away. In addition, all states in $\mathcal{T}_G$ are at least η_1 distance away from each other. This implies that there is a bound on the density of states. Using this intuition, the problem of estimating the maximum number of neighbors of a state can be restated as a sphere packing problem (Conway and Sloane, 1999).

Let x be the state added to $\mathcal{T}_G$ and S_1 and S_2 be two spheres centered at x and with radii η_1 and η_2, respectively. Each neighbor of x can be thought of as a sphere with radius $\eta_1/2$ and center belonging to the volume delimited by the two spheres S_1 and S_2. As $\eta_1 < \eta_2 < c\eta_1$, for some $c > 1$, it follows that there will be only a finite number of spheres that can be placed inside the described volume. Let N_S be the number of spheres, then a conservative upper bound is given by the following ratio

$$N_S \leqslant \frac{V(\eta_2) - V(\eta_1)}{V(\frac{\eta_2}{2})} \leqslant \frac{V(c\eta_2) - V(\eta_1)}{V(\frac{\eta_2}{2})} = 2^d(c^d - 1) \leqslant 2^{(d + \log_2 c)}$$

where d is the dimension of the configuration space $\mathcal{C}$ and $V(\alpha)$ is the volume of a d-sphere of radius $\alpha \geqslant 0$. Thus, x has at most $O(1)$ neighbors. This implies that Algorithm 1 adds at most $O(1)$ transitions to $\mathcal{T}_G$ when adding a new state x. As $\mathcal{T}_G$ is a sparse graph before adding the state x, it follows that $\mathcal{T}_G$ will remain a sparse graph. □

Remark 9. *Note that the exact value of N_S may depend not only on the dimension d of the configuration space $\mathcal{C}$, but also on the shape of $\mathcal{C}$ if x is close to its boundary.*

The number N_S is closely related to the kissing number (Conway and Sloane, 1999) in dimension d. The kissing number τ_d is the maximum number of non-overlapping d-spheres that touch another given d-sphere. It is easy to see that τ_d is a lower bound for the maximum value of N_S. In Conway and Sloane (1999), a linear optimization procedure to compute an upper bound for any dimension is presented. It is also known (Talata, 1998) that τ_d is exponential in d, i.e., $\tau_d \geqslant 2^{\alpha d}$, where $\alpha > 0$ is a constant. Thus, the maximum value of N_S is of order 2^d.

Haeupler et al. (2012) showed that any incremental algorithm that maintain a topological order and satisfies a "locality" property must take at least $\Omega(n\sqrt{m})$ time, where n is the number of nodes in the graph and m is the number of edges. The "locality" property is a mild assumption that restricts the algorithm to reorder only vertices that are affected by the addition of an edge. A vertex x is affected by the additional edge u, v if there is another vertex y such that $x < y$ in the original topological ordering, but must be changed to $x > y$. For more details see Haeupler et al. (2012). However, it is conjectured (Haeupler et al., 2012) that this bound holds in general (Conjecture 10).

In the following, we assume that $m = \Omega(n)$. In addition, to simplify the exposition, we assume without loss of generality that initially $\mathcal{T}_G$ has all n states and no transitions. This assumption is not restrictive, because vertex addition takes only $O(1)$.

Conjecture 10. *Any incremental cycle detection algorithm takes at least $\Omega(n\sqrt{m})$ time, where n is the number of vertices the graph and m is the number of edges added to it.*

Theorem 11. *If Conjecture 10 is true, then the complexity of any incremental checking algorithm for satisfying paths in a given transition system $\mathcal{T}_G$ is at least $\Omega(n\sqrt{m})$, where $n = |\mathcal{T}_G|$ and m is the number of transitions added to $\mathcal{T}_G$.*

Proof. Let $\mathcal{T}_G$ be a transition system with n states and m transitions, $\Delta_{\mathcal{T}_G} = \{tr_1, \ldots, tr_m\}$. In the following, we consider algorithms that return true or false whether adding a

given transition to a transition system $\mathcal{T}_G$ yields a satisfying run or not with respect to a given specification Φ_G. Let $A(\mathcal{T}_G, \Phi_G)$ be an incremental checking algorithm. We want to show that any such incremental algorithm takes at least $\Omega(n\sqrt{m})$ time.

It is well known (Baier and Katoen, 2008) that for any ω-regular language L there is a corresponding nondeterministic Büchi automaton, which accepts all and only the (infinite) words of L. As such, any encoding of the specification (LTL, CTL, CTL*, μ-calculus, etc.) has a corresponding Büchi automaton. Let $\mathcal{B}$ be the Büchi automaton corresponding to the ω-regular specification and $\bar{\mathcal{P}}_G = \mathcal{T}_G \times \mathcal{B}$ be the full product automaton without the reachability requirement.

Assume without loss of generality that the first $m-1$ transitions of $\mathcal{T}_G$ do not induce a satisfying run. Thus, only the mth transition may induce a satisfying run. Note, that the assumption is not limiting, because after a satisfying run is detected any additional transition will not change the result.

Let $\mathcal{P}_G^0, \ldots, \mathcal{P}_G^m$ be a sequence of subgraphs of $\bar{\mathcal{P}}_G$ with the following properties:

1. $\mathcal{P}_G^0, \ldots, \mathcal{P}_G^{m-1}$ are acyclic;
2. $\mathcal{P}_G^m$ is cyclic if and only if there is a satisfying run in $\mathcal{T}_G$ with respect to Φ_G;
3. $\emptyset = \Delta_{\mathcal{P}_G^0} \subseteq \Delta_{\mathcal{P}_G^1} \subseteq \ldots \subseteq \Delta_{\mathcal{P}_G^m}$;
4. $m' = \left|\Delta_{\mathcal{P}_G^m}\right| = \Omega(m)$.

It follows that procedure A solves the incremental cycle detection problem for $\mathcal{P}_G^0, \ldots, \mathcal{P}_G^m$. Therefore, A must take at least $\Omega(n'\sqrt{m'}) = \Omega(n\sqrt{m})$.

To complete the proof, we must show that there exists a subsequence $(\mathcal{P}_G^i)_{0 \leqslant i \leqslant m}$ for a given $\bar{\mathcal{P}}_G$ and a sequence $\{tr_1, \ldots, tr_m\}$ of transitions of $\mathcal{T}_G$. We define the subgraphs recursively as follows: (1) $\mathcal{P}_G^m$ is the maximum acyclic spanning subgraph of $\bar{\mathcal{P}}_G$ if $\mathcal{T}_G$ does not contain a satisfying run or $\mathcal{P}_G^m = \bar{\mathcal{P}}_G$ otherwise; (2) $\mathcal{P}_G^i$ is the maximum acyclic spanning subgraph of $\mathcal{P}_G^{i+1}|_{E_i}$ for all $i \in \{0, \ldots, m-1\}$, where $E_i = \{tr_1, \ldots, tr_i\} \times \delta_{\mathcal{B}}$ and $\mathcal{P}_G^{i+1}|_{E_i}$ is the subgraph of $\mathcal{P}_G^{i+1}$ with transitions restricted to E_i. From the definition it immediately follows that $\Delta_{\mathcal{P}_G^i} \subseteq (E_i \cap \Delta_{\mathcal{P}_G^{i+1}}) \subseteq \Delta_{\mathcal{P}_G^{i+1}}$ for all $0 \leqslant i \leqslant m-1$ and $\Delta_{\mathcal{P}_G^0} = \emptyset$. Thus, by construction conditions (1), (2), and (3) are satisfied. The last requirement is trivially true when $\mathcal{T}_G$ contains a satisfying run. In addition, when $\mathcal{T}_G$ does not contain a satisfying run, then the maximum acyclic graph of $\bar{\mathcal{P}}_G$ retains at least half the transitions. Any digraph G may be decomposed into two acyclic subgraphs (Wood, 2004) such that their edge sets form a partition of the edge set of G. It follows that at least one (acyclic) subgraph has half of the edges of G. Thus, we have that $m' = \Omega(m)$. □

Remark 12. *Note that Theorem 7 gives a lower bound for all incremental checking procedures with respect to the number of states and transitions that are added.*

The following corollaries of Theorem 11 are easy to prove.

Corollary 13. *If Conjecture 10 is true, then Algorithm 2 has the best possible complexity for transition systems that are sparse graphs.*

Corollary 14. *Algorithm 2 has the best possible complexity for transition systems, which are sparse graphs, among all incremental algorithms with the "locality" property.*

5.1.5. Probabilistic completeness. The presented RRG-based algorithm retains the probabilistic completeness of RRG (Karaman and Frazzoli, 2011).

Theorem 15. *Algorithm 1 is probabilistically complete.*

Proof. We start by noting that any word in a ω-regular language can be represented by a finite state Büchi automaton (Baier and Katoen, 2008). This is important, because this shows that a solution, represented by a transition system, is completely characterized by a finite number of states. Let us denote by $\bar{X}$ the finite set of states that define a solution. It follows from the way regions are defined that we can choose a neighborhood around each state in $\bar{X}$ such that the system can be steered in one step from all points in one neighborhood to all points in the next neighborhood. Thus, we can use induction to show that (Karaman and Frazzoli, 2012): (1) there is a non-zero probability that a sample will be generated inside the neighborhood of the first state in the solution sequence; (2) if there is a state in $\mathcal{T}$ that is inside the neighborhood of the kth state from the solution sequence, then there is a non-zero probability that a sample will be generated inside the $k+1$th state's neighborhood. Therefore, as the number of samples goes to infinity, the probability that the transition system $\mathcal{T}$ has nodes belonging to all neighborhoods of states in $\bar{X}$ goes to 1. To finish the proof, note that we have to show that the algorithm is always able to generate samples with the desired "sparseness" property. However, recall that the bound functions must converge to 0 (as the number of states goes to infinity) fast enough such that the set of configurations for which "far" function returns a non-empty list has non-zero measure with respect to the sampling distribution. This concludes the proof. □

5.2. On-line algorithm

The approach for solving the on-line part of the planning problem is based on the RRT algorithm, a probabilistically complete sampling-based path planning method. We modify the standard RRT in order to find local paths that preserve the satisfaction of the global specification Φ_G, while servicing on-line requests and avoiding locally sensed obstacles.

To keep track of validity of samples (random configurations) with respect to the global specification Φ_G, we propose a method that combines the ideas presented in Bauer et al. (2007) on monitors for LTL formulae and Ding et al. (2014) on potential functions. The problem considered in Bauer et al. (2007) is to decide as soon as possible whether

a given (infinite) word w satisfies a LTL formula ϕ. The main idea is to keep track of Büchi states corresponding to a finite prefix of w with respect to both ϕ and $\neg\phi$ concurrently. If one of the two sets of Büchi states corresponding to ϕ or $\neg\phi$ becomes empty, then we can conclude that the specification is either violated or satisfied. If both sets are non-empty, then nothing can be said about $w \models \phi$. In our case, we just use half of a monitor, because we are interested only in checking whether steering the robot to new samples violates Φ_G. The potential functions approach described in Ding et al. (2014) is used to address the problem of connecting the locally generated path to states in the global transition system such that Φ_G is satisfied.

5.2.1. Potential functions. Ding et al. (2014) defined a potential function over the states of the product automaton between a transition system and a Büchi automaton. The potential function captures the distance from each state of the product to the closest final state. It can be thought of as a distance to satisfaction and resembles a Lyapunov function. We extend this notion to define potential functions on the states of the global transition system. This extension allows us to reason about the change of potential between nodes of $\mathcal{T}_G$ connected through local paths instead of a direct transition. The local paths are generated as branches of a tree by the proposed RRT-based algorithm. The definitions of self-reachable set and potential function for product automaton states presented in the following are adapted from Ding et al. (2014).

Let $\mathcal{P}_G = \mathcal{T}_G \times \mathcal{B} = (S_{\mathcal{P}_G}, S_{\mathcal{P}_{G0}}, \Delta_{\mathcal{P}_G}, \omega_{\mathcal{P}_G}, F_{\mathcal{P}_G})$ be a product automaton between a transition system $\mathcal{T}_G$ and Büchi automaton $\mathcal{B}$. We denote by $\mathcal{D}(p,p')$ the set of all finite trajectories from a state $p \in S_{\mathcal{P}_G}$ to a state $p' \in S_{\mathcal{P}_G}$:

$$\mathcal{D}(p,p') = \{p_1 \dots p_n | p_1 = p, p_n = p'; p_k \rightarrow_{\mathcal{P}_G} p_{k+1} \forall k = 1, \dots, n-1; \forall n \geqslant 2\} \quad (2)$$

A state $p \in S_{\mathcal{P}_G}$ is said to reach a state $p' \in S_{\mathcal{P}_G}$ if $\mathcal{D}(p,p') \neq \emptyset$. The length of a path is defined as the sum of the weights corresponding to the transitions it is composed of

$$L(\mathbf{p}) = \sum_{k=1}^{n-1} \omega_{\mathcal{P}_G}(p_k, p_{k+1}) \quad (3)$$

For $p, p' \in S_{\mathcal{P}_G}$, the distance between p and p' is defined as follows:

$$d(p,p') = \begin{cases} \min_{\mathbf{p} \in \mathcal{D}(p,p')}(L(\mathbf{p})) & \text{if } \mathcal{D}(p,p') \neq \emptyset \\ \infty & \text{if } \mathcal{D}(p,p') = \emptyset \end{cases} \quad (4)$$

The weight function $\omega_{\mathcal{P}_G}$ is positive, because it is induced by the distance of the underlying (metric) space. This implies (Ding et al., 2014) that $d(p,p') \geqslant 0$ for all $p, p' \in S_{\mathcal{P}_G}$ with equality if and only if $p = p'$, because we consider only LTL$_{-X}$ specifications. This means that there can be no progress towards the global specification, i.e., change in the Büchi automata states, without the robot visiting a region different than the current one it is in.

A set $A \subset S_{\mathcal{P}_G}$ is *self-reachable* if and only if all states in A can reach a state in A. Formally, a set A is self-reachable if for all $p \in A$ there is a state $p' \in A$ such that $\mathcal{D}(p,p') \neq \emptyset$.

Definition 16. (Potential function of states in $\mathcal{P}_G$). *The potential function $V_{\mathcal{P}_G}(p)$, $p \in S_{\mathcal{P}_G}$ is defined as*

$$V_{\mathcal{P}_G}(p) = \begin{cases} \min_{p' \in F^*_{\mathcal{P}_G}} d(p,p') & \text{if } p \notin F^*_{\mathcal{P}_G} \\ 0 & \text{if } p \in F^*_{\mathcal{P}_G} \end{cases} \quad (5)$$

where $F^*_{\mathcal{P}_G} \subset F_{\mathcal{P}_G}$ is the maximal self-reachable set of final states of $\mathcal{P}_G$.

The potential function is non-negative for all states of $\mathcal{P}_G$. It is zero for some $p \in S_{\mathcal{P}_G}$ if and only if p is a final state and p can reach itself or a self-reachable final state. In addition, if $V_{\mathcal{P}_G}(p) = \infty$, $p \in S_{\mathcal{P}_G}$, then p does not reach any self-reachable final states.

Definition 17. (Potential function of states in $\mathcal{T}_G$). *Let $x \in X$ and $B \subseteq \beta_{\mathcal{P}_G}(x)$. The potential function of x with respect to B is defined as*

$$V_{\mathcal{T}_G}(x, B) = \min_{s \in B} V_{\mathcal{P}_G}((x,s)) \quad (6)$$

In addition, the minimum potential of x is defined as $V^*_{\mathcal{T}_G}(x) = V_{\mathcal{T}_G}(x, \beta_{\mathcal{P}_G}(x))$.

The minimum potential of a state x of $\mathcal{T}_G$ is the minimum potential of all states in $\mathcal{P}_G$ that correspond to x. The actual potential is defined to capture the fact that not all Büchi states may be available in order to achieve the minimum potential.

Ding et al. (2014) presented an algorithm to compute the potential function $V_{\mathcal{P}_G}(\cdot)$ over the states of the product automaton. The complexity of the algorithm is $O(|F_{\mathcal{P}_G}|^3 + |F_{\mathcal{P}_G}|^2 + |S_{\mathcal{P}_G}|^2 \times |F_{\mathcal{P}_G}|)$ (Ding et al., 2014).

We propose an improved algorithm (see Algorithm 3), which reduces the complexity by a polynomial factor.

Theorem 18. *Algorithm 3 correctly computes the potential function $V_{\mathcal{P}_G}(\cdot)$ for a given product automaton $\mathcal{P}_G$ and its complexity is $O(|S_{\mathcal{P}_G}| \log |S_{\mathcal{P}_G}| + |\Delta_{\mathcal{P}_G}|)$.*

Remark 19. *In the proposed framework, the computation of the SCC in Algorithm 3 (lines 1–2) may be skipped, because the off-line planning Algorithm 1 already maintains SCCs of $\mathcal{P}_G$. Thus, Algorithm 3 is better suited for use in conjunction with the off-line algorithms.*

Proof. The improvement achieved by Algorithm 3 is based on two observations: (1) if the maximal self-reachable final states set $F^*_{\mathcal{P}_G}$ is known, then the potential function $V_{\mathcal{P}_G}(\cdot)$ can be computed by running Dijkstra's algorithm once instead of $|F_{\mathcal{P}_G}|$ times as in Ding et al. (2014); (2) self-reachability is a property about the existence of cycles in $\mathcal{P}_G$ and can therefore be inferred from the SCC directed acyclic graph (DAG) of $\mathcal{P}_G$.

Algorithm 3: Compute Potential Function $V_{\mathcal{P}_G}(\cdot)$

Input: $\mathcal{P}_G$– product automaton
Output: Boolean value indicating whether there are self-reachable final states
1 $\mathcal{P}_G \leftarrow \mathcal{P}_G \cup (\{v\}, \{(v,p,0) : p \in S_{\mathcal{P}_G}\})$ // add virtual state v and connect it to all states
2 $scc, dag \leftarrow StronglyConnectedComponentsDAG(\mathcal{P}_G)$ // compute SCC DAG for $\mathcal{P}_G$
3 $scc_v \leftarrow \{v\}$
4 $F^*_{\mathcal{P}_G} \leftarrow ComputeSRFS(dag, F_{\mathcal{P}_G}, scc_v)$ // compute self-reachable final states
5 $\mathcal{P}_G \leftarrow \mathcal{P}_G \backslash \{v\}$ // remove virtual state v and all its incident transitions
6 **if** $F^*_{\mathcal{P}_G} = \emptyset$ **then** // if there are no self-reachable final states
7 **return** False
8 $\mathcal{P}_G \leftarrow \mathcal{P}_G \cup (\{v\}, \{(p,v,0) : p \in F^*_{\mathcal{P}_G}\})$ // add virtual state v and connect all self-reachable final states to it with weight 0
9 $V_{\mathcal{P}_G} \leftarrow ReverseDijkstra(\mathcal{P}_G, sink = v)$ // compute potentials for each state with v as sink
10 $\mathcal{P}_G \leftarrow \mathcal{P}_G \backslash \{v\}$ // remove virtual state v and all its incident transitions
11 **return** True

Algorithm 3 computes the potential function $V_{\mathcal{P}_G}(\cdot)$ by first computing $F^*_{\mathcal{P}_G}$ (lines 1–5) using the SCC DAG (line 2) and Algorithm 4. However, Algorithm 4 performs a depth-first search (DFS) of *dag* starting from a given SCC scc_r. Thus, it returns only the states of $F^*_{\mathcal{P}_G}$ that belong to scc_r and its descendants. In order to avoid calling Algorithm 4 for all SCC, we add a virtual node v to $\mathcal{P}_G$ (line 1), which is connected to all states of $\mathcal{P}_G$, and then compute the SCC DAG. Because v only has outgoing transitions, it cannot belong to any cycle. Thus, the SCC scc_v containing v is a singleton and is connected to all other SCCs in *dag*. It follows that running Algorithm 4 on *dag* with starting SCC scc_v (line 4) correctly computes $F^*_{\mathcal{P}_G}$. Afterwards, v and all its incident transitions are removed from $\mathcal{P}_G$ (line 5). If there are self-reachable final states (line 6), then the algorithm proceeds to compute the potentials using Dijkstra's algorithm starting from $F^*_{\mathcal{P}_G}$ and traversing transitions in the opposite direction (lines 8–10), i.e., using the incoming transitions instead of the outgoing transitions. Again, in order to avoid calling Dijkstra's algorithm for every state in $F^*_{\mathcal{P}_G}$, we add a virtual node v to $\mathcal{P}_G$. All states in $F^*_{\mathcal{P}_G}$ are connected to v with weight 0. Because v has only ingoing transitions and $\omega_{\mathcal{P}_G}((p,p'))>0$ for all $(p,p') \in \Delta_{\mathcal{P}_G}$, it follows that v does not belong to any cycles and Dijkstra's algorithm correctly computes the potential function for every $p \in S_{\mathcal{P}_G}$:

$$
\begin{aligned}
d(p,v) &= \begin{cases} \min_{p' \in F^*_{\mathcal{P}_G}} \{d(p,p') + \omega_{\mathcal{P}_G}((p',v))\} & \text{if } p \notin F^*_{\mathcal{P}_G} \\ \omega_{\mathcal{P}_G}((p',v)) & \text{if } p \in F^*_{\mathcal{P}_G} \end{cases} \\
&= \begin{cases} \min_{p' \in F^*_{\mathcal{P}_G}} d(p,p') & \text{if } p \notin F^*_{\mathcal{P}_G} \\ 0 & \text{if } p \in F^*_{\mathcal{P}_G} \end{cases} \\
&= V_{\mathcal{P}_G}(p)
\end{aligned}
$$

Algorithm 4: Compute Self-Reachable Final States – computeSRFS()

Input: *dag*– the SCC directed acyclic graph of $\mathcal{P}_G$
Input: $F_{\mathcal{P}_G}$– the set of final states of $\mathcal{P}_G$
Input: scc_r– the current root SCC used by the DFS algorithm
Output: *srfs*– the set of self-reachable final states
1 $srfs \leftarrow \emptyset$
2 $visited(scc_r) \leftarrow True$
3 **foreach** $scc_n \in dag.out(scc_r)$ **do**
4 **if** $\neg visited(scc_n)$ **then**
5 $srfs \leftarrow srfs \cup computeSRFS(dag, F_{\mathcal{P}_G}, scc_n)$
6 **else**
7 $srfs \leftarrow srfs \cup S(scc_n)$
8 **if** $|scc_r|>1 \vee srs \neq \emptyset \vee (scc_r = \{p\} \wedge (p,p) \in \Delta_{\mathcal{P}_G})$ **then**
9 $srfs \leftarrow srfs \cup (F_{\mathcal{P}_G} \cap scc_r)$
10 $S(scc_r) \leftarrow srfs$
11 **return** *srfs*

Algorithm 5: Tracking Büchi States of Local Samples

Input: $\mathcal{B}$– Büchi automaton corresponding to Φ_G
Input: $w = \sigma_1 \ldots \sigma_n$– a finite word over 2^{Π_G}
Input: B– a set of Büchi states from which the tracking starts
Output: B_f– set of Büchi states available after the last symbol of w
1 $B_f \leftarrow B$
2 **for** $k \leftarrow 1 \ldots (n-1)$ **do**
3 $B' \leftarrow \emptyset$
4 **foreach** $s \in B_f$ **do**
5 $B' \leftarrow B' \cup \{s' \in S_{\mathcal{B}} | (s,s') \in \Delta_{\mathcal{B}}\}$
6 $B_f \leftarrow B'$
7 **return** B_f

The analysis presented above relies on the fact that Algorithm 4 correctly computes $F^*_{\mathcal{P}_G}$. In the following, we prove by structural induction with respect to *dag* that Algorithm 4 correctly computes $F^*(scc_r)$, where scc_r is an SCC of *dag* and $F^*(scc_r)$ is the maximal subset of $F^*_{\mathcal{P}_G}$ whose states belong to scc_r and its descendants.

First, note that by definition a final state p_f belongs to $F^*_{\mathcal{P}_G}$ if and only if: (1) p_f belongs to a cycle or equivalently to a SCC of $\mathcal{P}_G$ with more than one state; (2) p_f has a self-loop; or (3) p_f reaches another state in $F^*_{\mathcal{P}_G}$. As *dag* is acyclic it follows that condition (3) can be reduced to checking whether $F * (scc_n)$ is non-empty for some successor scc_n of scc_r. This implies that $F^*(scc_r)$ is unique for every scc_r and it can be computed recursively using DFS. The recursive algorithm starts by marking the current SCC scc_r as visited (line 2) and proceeds to compute the union of $F^*()$ for all successors of scc_r (lines 3–7). If a successor scc_n was not visited previously, then the procedure is called recursively starting from scc_n (lines 4–5), otherwise the stored set corresponding to scc_n is used (line 7). The next step is to add the self-reachable final states of scc_r to *srfs* (lines 8–9). The *srfs* is stored in $S(scc_r)$ for possible later use (line 10). We need to show that $S(scc_r) = F^*(scc_r)$, for all scc_r in *dag*.

Algorithm 6: Planning Algorithm

Input: Φ_G– the global LTL$_{-X}$ specification
Input: *prio*– the priority function for on-line requests
Input: x_0 initial configuration of the robot
1 Convert Φ_G into Büchi automaton $\mathcal{B}$
2 Compute $\mathcal{T}_G$ and $\mathcal{P}_G = \mathcal{T}_G \times \mathcal{B}$ starting at x_0 using Algorithm 1
3 Compute potential function $V_{\mathcal{P}_G}(\cdot)$
4 $path \leftarrow emptyList()$
5 $x_c \leftarrow x_0$
6 $B(x_c) \leftarrow \beta_{\mathcal{P}_G}(x_c)$
7 **while** *True* **do**
8 $I \leftarrow getLocalRequests()$
9 **if** $checkPath(I, path) \vee \neg path.hasNext()$ **then**
10 $path \leftarrow planLocally(x_c, \mathcal{P}_G, \mathcal{B}, prior, I)$
11 $x_n \leftarrow path.next()$
12 $enforce(x_c \rightarrow x_n)$
13 $x_c \leftarrow x_n$

The *base case* is trivial, because it involves the SCCs without any outgoing transitions in *dag*. It follows that *srfs* at line 8 is empty. In addition, all final states in scc_r satisfying conditions (1) or (2) are added to *srfs*. Thus, $S(scc_r) = F^*(scc_r)$.

For the *induction step*, we assume that Algorithm 4 correctly computes $F^*(scc_n)$ for all successors of scc_r (line 5). Note that if a successor scc_n was already visited at some previous step, Algorithm 4 was called with scc_n as starting SCC. Therefore, $S(scc_n)$ (line 7) is assumed to be computed correctly by the induction hypothesis. As in the base case, if either condition (1) or (2) hold, then Algorithm 4 adds all final states in scc_r to *srs* and it follows that $S(scc_r) = F^*(scc_r)$. The remaining case is when scc_r is a singleton $\{p\}$ and p has no self-loop. In this case, $p \in F^*(scc_r)$ if and only if p reaches some other state in $F^*(scc_r)$. As *dag* is acyclic, p can only reach states in the descendants SCC of scc_r. On the other hand, by the induction hypothesis we have that $srfs = F^*(scc_r) \setminus \{p\}$ at line 8. It follows that p is added to $S(scc_r)$ if *srfs* is non-empty at line 8. Thus, we have $S(scc_r) = F^*(scc_r)$ in this case as well when Algorithm 4 returns.

The complexity of Algorithm 3 is $O(|S_{\mathcal{P}_G}| \log |S_{\mathcal{P}_G}| + |\Delta_{\mathcal{P}_G}|)$. It is easy to see that the operations on lines 1, 5, 8, and 10 take $O(|S_{\mathcal{P}_G}|)$, while computing the SCC DAG (line 2) and Dijkstra's algorithm (line 9) have $O(|\Delta_{\mathcal{P}_G}|)$ and $O(|S_{\mathcal{P}_G}| \log |S_{\mathcal{P}_G}| + |\Delta_{\mathcal{P}_G}|)$ complexity, respectively. In addition, computing $F^*_{\mathcal{P}_G}$ using Algorithm 4 takes at most $O(|S_{\mathcal{P}_G}| + |\Delta_{\mathcal{P}_G}|)$. The SCC DAG graph *dag* has at most the same number of states and transitions as $\mathcal{P}_G$. Algorithm 4 is also a DFS and each SCC is processed once and each transition of *dag* is traversed once. Therefore, the overall complexity of processing the SCCs in Algorithm 4 (lines 8–10) is linear in the number of states of $\mathcal{P}_G$. Adding the complexity of all steps, we obtained the stated complexity bound. □

5.2.2. Satisfying local paths with respect to Φ_G. Local paths in our RRT-based algorithm connect states of the global transition system. Let $x, x' \in \mathcal{T}_G$ and $\mathbf{x} = x_1 \ldots x_n$ be a local path connecting $x_1 = x$ and $x_n = x'$ and $\mathbf{o} = o_1 \ldots o_n$ be the output trajectory corresponding to $\mathbf{x}$ with respect to the global proprieties ($o_k \in 2^{\Pi_G}, \forall k = 1 \ldots n$). We need to ensure that there is a satisfying run in $\mathcal{T}_G$ starting at x' after traversing $\mathbf{x}$. Thus, we need to consider two problems: (1) how to keep track of available Büchi states as local samples are generated and (2) how to connect a local path's endpoint (tree leaf) to the global transition system $\mathcal{T}_G$.

The first problem is solved by Algorithm 5, which determines the set of Büchi states given a word w over 2^{Π_G}. Algorithm 5 solves this problem by repeatedly computing the set of outgoing neighboring states of $\mathcal{B}$ for all states in the previous iteration. To check whether a local path can be connected to the state $x_n = x' \in X$, we just need to verify that it has finite potential, i.e., $V_{\mathcal{T}_G}(x', B) < \infty$, where B is the set of available Büchi states after traversing $\mathbf{x}$, in this case $w = \mathbf{o}$.

The second problem has a simple solution in this setting. We choose the state in $\mathcal{T}_G$ which has (finite) minimum potential after traversing a branch of the RRT tree. In addition, the line segment between the leaf state from the tree and the state in $\mathcal{T}_G$ must be collision free (see Section 5.2.3).

5.2.3. On-line planning algorithm. The overall planning algorithm, outlined in Algorithm 6, is composed of the off-line preprocessing steps of computing the global transition system $\mathcal{T}_G$, the product automaton $\mathcal{P}_G = \mathcal{T}_G \times \mathcal{B}$ and the potential function for $\mathcal{P}_G$ and the on-line loop. At each step of the loop, the robot scans for local requests and obstacles and checks whether it needs to compute a new local path. Re-planning is performed in four cases: (1) if the current path is empty; (2) a higher-priority request was detected; (3) the chosen request disappeared; and (4) the local path collides with a local obstacle. Büchi states are tracked starting from the initial configuration of the robot, corresponding to the initial state of $\mathcal{T}_G$. Map B is used to store the tracked Büchi states. Figure 4 shows how a the local planning algorithm interacts with the $\mathcal{T}_G$ and locally sensed requests and obstacles.

The local path planning algorithm is shown in Algorithm 7 and is based on RRT. The procedure incrementally constructs the local transition system $\mathcal{T}_L$. The initial (root) state of $\mathcal{T}_L$ is the current configuration of the robot x_c. The map *serv* indicates whether a state or any of its ancestors serviced the on-line request with the highest priority. If there are no requests, then *serv* is true for all states of $\mathcal{T}_L$.

The construction of the RRT proceeds by generating a new random sample (line 4) inside the sensing area of the robot, steer the system towards it (lines 5–6) and checking whether it is a valid state (lines 8–9). Samples are generated such that their images in $\mathcal{D}$ belong to the sensing area of the robot. The *nearest* function (line 5) is a standard RRT

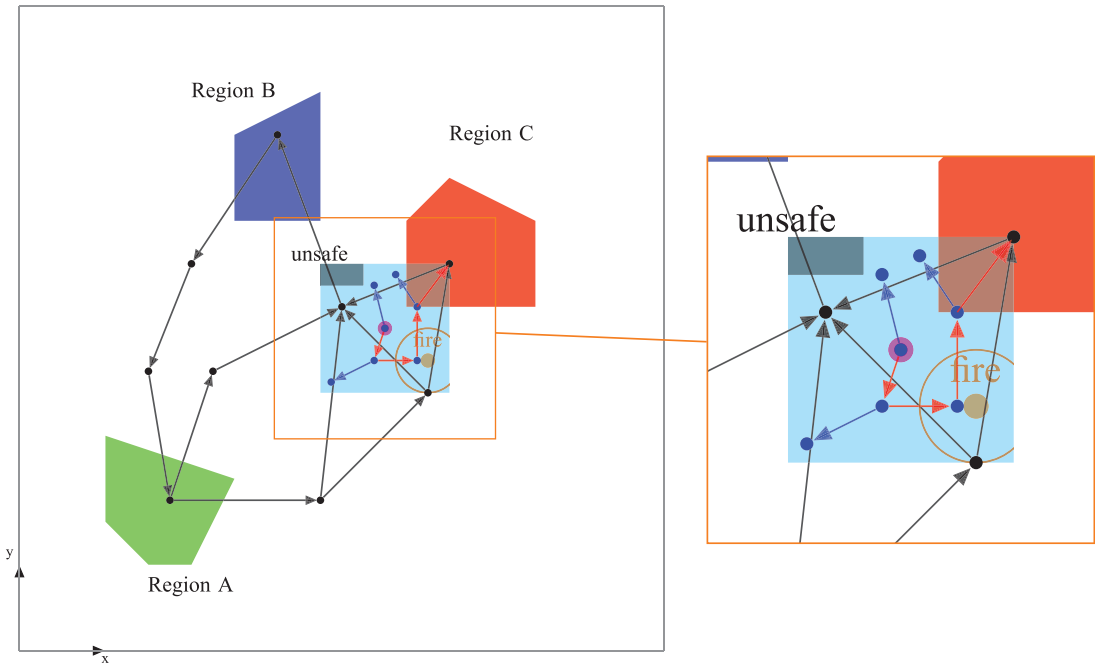


Fig. 4. The same environment as in Figure 2, but also showing the global transition system $\mathcal{T}_G$ (in black) and the local transition system $\mathcal{T}_L$ (in blue and red). The robot's current position x_c is marked by the magenta disk and coincides with the root of $\mathcal{T}_L$. The sensing area is again in cyan and a *fire* request and a local obstacle (*unsafe*) are detected. Note that in this figure only the portion of the *unsafe* area that is inside the sensing area is detected. In addition, the *survivor* request is not detected at all. The local control strategy, which corresponds to a path from x_c to a leaf and then to a state in $\mathcal{T}_G$, was found and is shown in red. The last transition of the local path is the link between $\mathcal{T}_L$ and $\mathcal{T}_G$. This local path satisfies the global and local mission specification described in Example 5.
Note: Colour version of the figure is available online.

primitive that returns the nearest state in $\mathcal{T}_L$ based on the distance function associated with $\mathcal{C}$. We assume that we have access to a *steer* function (see Section 5.1.1), which drives the system from x_n to a configuration $x \in \mathcal{C}$, where x is the closest configuration to the new sample x_s and it is within η_L distance away from x_n (Karaman and Frazzoli, 2011; LaValle and Kuffner, 2001). The *label* primitive function (line 7) is used to annotate x with the global properties it satisfies. The new state x is valid if its corresponding set of Büchi states is non-empty and the line segment from its parent x_n to itself is a simple collision-free line segment. Algorithm 5 is used to compute the set of available Büchi states for x. The primitive function *isSimpleSegment* is used to ensure that the set of global properties along the potential new transition (x_n, x) changes at most once (see Section 5.1.1). The *collisionFree* primitive is used to check whether the image in the workspace of the line segment $(x_n, x) \in \mathcal{C}$ collides with a local obstacle in $\mathcal{D}$. If these tests are passed, the procedure adds the state x and the transition (x_n, x) to $\mathcal{T}_L$ (lines 11–12). In addition, the *serv* map is updated by checking if either the parent state x_n (or some ancestor) or the state itself x has serviced the selected on-line request.

We also require that the state x_G of $\mathcal{T}_G$ has a lower (actual) potential than the last visited state $x_{G'}$ of $\mathcal{T}_G$. This condition is not enforced, if the potential of $x_{G'}$ is zero, but we still require $x_G \neq x_{G'}$.

Algorithm 7: Local Path Planning

Input: x_c– current configuration of the robot
Input: $\mathcal{P}_G$– the product automaton $\mathcal{T}_G \times \mathcal{B}$
Input: $\mathcal{B}$– Büchi automaton corresponding to global specifications Φ_G
Input: *prior*– on-line requests priority function
Input: I– sensed requests and local obstacles
Output: *path*– computed local control strategy

```
1  Construct T_L = (X_L, x_c, Δ_L, ω_L, Π_L ∪ {π_O}, h_L) with x_c as initial state
2  serv(x_c) ← ¬I.hasRequest()
3  while ∄ x_c →*_{T_L} x_T → x_G w/ V_{T_G}(x_G, B(x_G)) < ∞ ∨ ¬serv(x_T) do
4      x_s ← generateSample(x_c, I.area)
5      x_n ← nearest(T_L, x_s)
6      x ← steer(x_n, x_s, η_L)
7      x ← label(x, I)
8      B(x) ← trackBuchiStates(B, h_L(x_n), B(x_n))
9      if B(x) ≠ ∅ ∧ isSimpleSegmnent(x_n, x)
       ∧ collisionFree(x_n, x) then
10         serv(x) ← serv(x_n) ∨ I.serviced(x, prior)
11         X_L ← X_L ∪ {x}
12         Δ_L ← Δ_L ∪ {(x_n, x)}
13 return x_c →*_{T_L} x_T → x_G
```

5.2.4. Correctness of local paths with respect to Φ_G

Theorem 20. *Let* $\mathbf{x} = x_1, \ldots$ *be an infinite path in* $\mathcal{C}$ *generated by Algorithm 6 and* $\mathbf{o} = o_1, \ldots$ *be the corresponding*

(infinite) output word generated by traversing $\mathbf{x}$. *If every call of Algorithm 7 finishes in finite time, then* $\mathbf{o}$ *satisfies the global specification* Φ_G, $\mathbf{o} \models \Phi_G$.

Proof. The condition that local path planning algorithm (Algorithm 7) always finishes in finite time implies that it was able to successfully find a local strategy every time the robot detected on-line requests and local obstacles. Therefore, this assumption implies that the environment is not adversary to the robot, i.e., it does not actively try to stop the robot from performing its mission.

By construction, every time Algorithm 7 finishes successfully it returns a local path which ends in a state x of $\mathcal{T}_G$ with finite (actual) potential. This implies that there is a state $p = (x, s)$ of $\mathcal{P}_G$ with finite potential, where $s \in B(x)$, and its potential is less than the potential of the previous state of $\mathcal{T}_G$ occurring in $\mathbf{x}$. As shown Ding et al. (2014), this guarantees that there is a state x' in $\mathbf{x}$ with zero potential and x' is a finite number of steps after x in $\mathbf{x}$.

By the hypothesis, $\mathbf{x}$ contains infinitely many states of $\mathcal{T}_G$ and an infinite number of them has zero potential. This concludes the proof, because the states with zero potential correspond to final Büchi states. □

Remark 21. *The complexity of the local path planning algorithm (Algorithm 7) is the same as for the standard RRT. The functions generateSample, steer, and nearest are standard primitives (Karaman and Frazzoli, 2011; LaValle and Kuffner, 2001). The functions label, isSimpleSegment, and collisionFree are primitives and check whether an on-line request was serviced and can be reduced to collision detection in the lower-dimensional workspace. Tracking Büchi states takes constant time (*$O(1)$*), because the global specification* Φ_G *is fixed.*

Remark 22. *The RRT-based Algorithm 7 is probabilistically complete at each procedure call, where the goal is to visit the servicing disk of the highest-priority request and the constraints are to avoid global regions that lead to violation of the global specification, avoiding local obstacles, sampling only within the sensing area, and leafs along servicing paths to connect to the global transition system. Thus, each problem instance reduces to the reach avoid problem with terminal cost. In this setting, RRT is probabilistically complete (Karaman and Frazzoli, 2011; LaValle and Kuffner, 2001). The meaning of completeness is not obvious in the setting of time-varying goals and constraints, and partial-information problems. Development of a theoretical framework about completeness for dynamic, partial information scenarios is beyond the scope of this article, and is left for future work.*

6. Case studies

In this section, we illustrate the usefulness and the performance of the proposed framework in simulated and experimental trials. The first part focuses on simulated trials where robots are given persistent tasks expressed as LTL formulae while servicing locally sensed requests. We show that the algorithms are able to handle planning in configuration spaces with up to 19 dimensions. In the second part, we present experiments on two robotic platforms: Cozmo and Baxter robots. The Cozmo is deployed in a labeled planner environment, and tasked to perform persistent surveillance missions while servicing requests detected using its on-board camera (see Section 6.2.1 for details of the experimental setup). In Baxter's case, persistent tasks, such as attending cooking pots, need to be performed using one six-degree-of-freedom (6-DoF) arm. Moreover, it needs to service requests associated with customers' plates (see Section 6.2.2 for the description of the setup).

The algorithms presented in this article are implemented in Python 2.7 and use the *LOMAP* (Ulusoy et al., 2013c) and *networkx* (Hagberg et al., 2008) libraries. The *spot* tool (Duret-Lutz et al., 2016) was used to convert the LTL specifications into Büchi automata.

The examples in Section 6.1 were ran on an Ubuntu 16.04 laptop with Intel Core i7 at 2.90 GHz and 32 GB of memory. The experiments in Section 6.2 were performed using an Ubuntu 14.04 desktop system with Intel Xeon 2.20 GHz processors and 64 GB of memory.

6.1. Simulations

In this section, we present an illustrative case study in a planar environment, and then focus on the scalability of the off-line and on-line algorithms with respect to the dimension of the robots' configuration spaces. Lastly, we compare the performance of the off-line algorithm against naïve approaches that do not use incremental and sparsity mechanisms, respectively.

6.1.1. Planar case. Consider the planar configuration space depicted in Figure 5 of a fully actuated robot, where the workspace coincides with the configuration space, and the submersion $\mathcal{H}$ is trivial. The initial configuration is $x_0 = (2, 2)$. The specification is to "*visit regions* $r1$, $r2$, $r3$ *and* $r4$ *infinitely many times while avoiding regions* $o1$, $o2$, $o3$ *and* $o4$." The corresponding LTL formula for the given mission specification is

$$\phi_1 = \square(\diamondsuit r1 \wedge \diamondsuit r2 \wedge \diamondsuit r3 \wedge \diamondsuit r4 \wedge \neg(o1 \vee o2 \vee o3 \vee o4)) \tag{7}$$

There are three local obstacles labeled *LO* and three dynamic requests: two *fire* requests and a *survivor* request. The three dynamic requests have a cyclic motion at a lower speed than that of the robot. The maximum distance traveled by the robot in one discrete time step is $\eta = 0.1$ (see the steer primitive in Algorithm 7, line 6). The priority function *prior* is defined such that *survivor* request have higher priority than *fire* request.

A solution with respect to the global specification of the problem is shown in Figure 5. A few iterations of Algorithm 1 are shown in Figure 6. The global satisfying

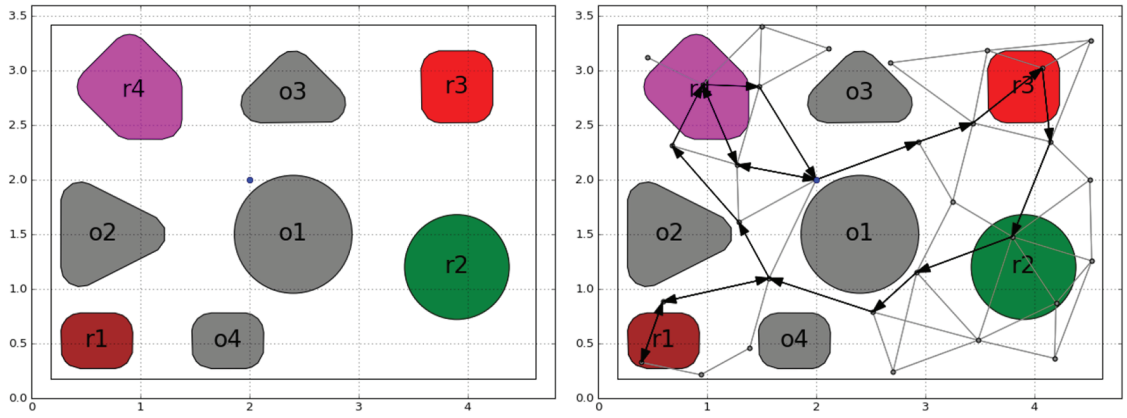
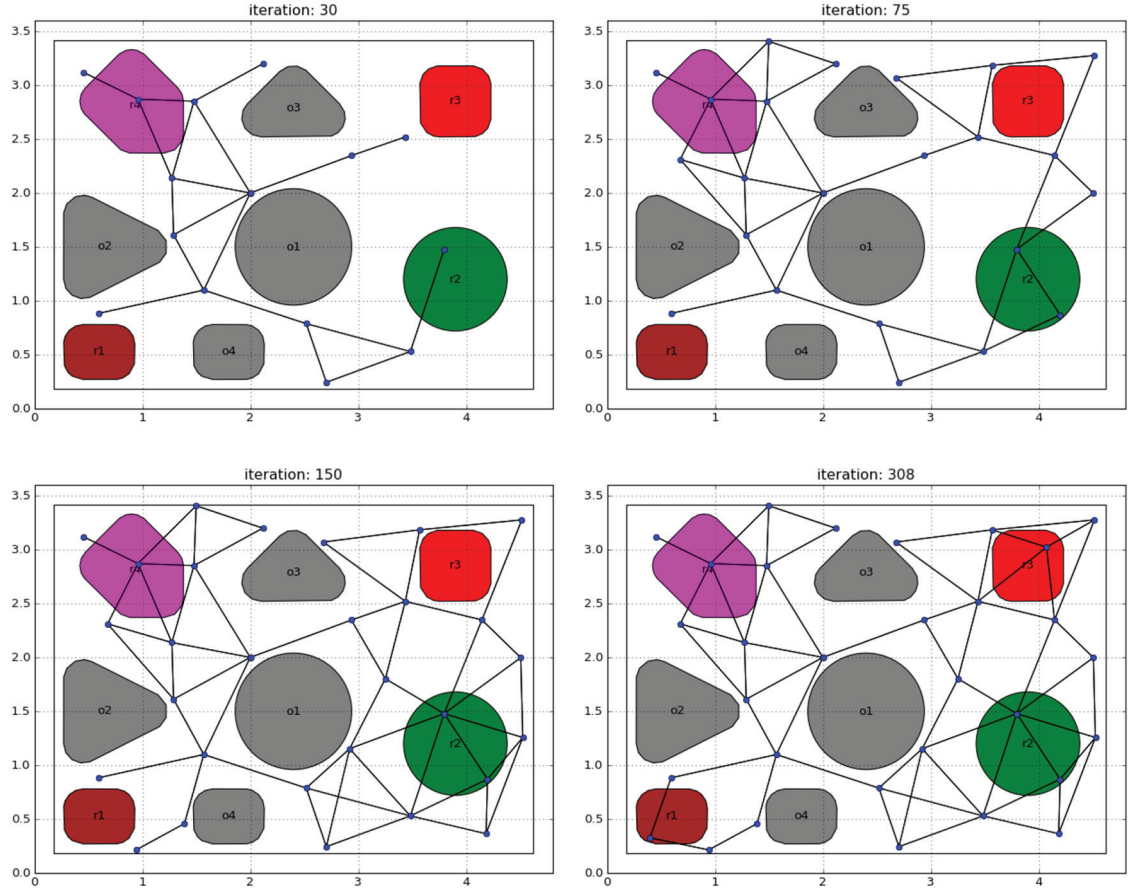


Fig. 5. One solution corresponding to the case in Section 6.1.1: the specification is to visit all the colored regions labeled $r1$ (brown), $r2$ (green), $r3$ (red), and $r4$ (magenta) infinitely often, while avoiding the dark gray obstacles labeled $o1$, $o2$, $o3$, and $o4$. The gray dots and arrows represent the 31 states and 116 transitions of the transition system $\mathcal{T}_G$, respectively. The starting configuration of the robot (the initial state of $\mathcal{T}_G$) is denoted by the blue dot. The black arrows represent the satisfying run (finite prefix, suffix pair) found by Algorithm 1. In this case, the prefix and suffix are composed of 15 and 18 states from $\mathcal{T}_G$, respectively.
Note: Colour version of the figure is available online.



Figu. 6. Transition systems obtained at earlier iterations corresponding to the solution shown in Figure 5 (to be read from left to right and top to bottom). The blue dots and black lines represent the states and transitions of $\mathcal{T}_G$, respectively.
Note: Colour version of the figure is available online.

path is given as a prefix–suffix pair. The prefix is executed once, and starts at the initial configuration x_0 and ends at configuration $x_{accept} = (0.42, 0.32)$ (lower left corner) associated with an accepting state $p_{accept} = (x_{accept}, s_{accept})$ in the product automaton $\mathcal{P}$. Recall that p_{accept} is an accepting state of $\mathcal{P}$ if and only if s_{accept} is an accepting state of the Buchi automaton $\mathcal{B}_{\phi_1}$. The suffix is the projection onto the global transition system $\mathcal{T}_G$ of a cycle in $\mathcal{P}$ starting and

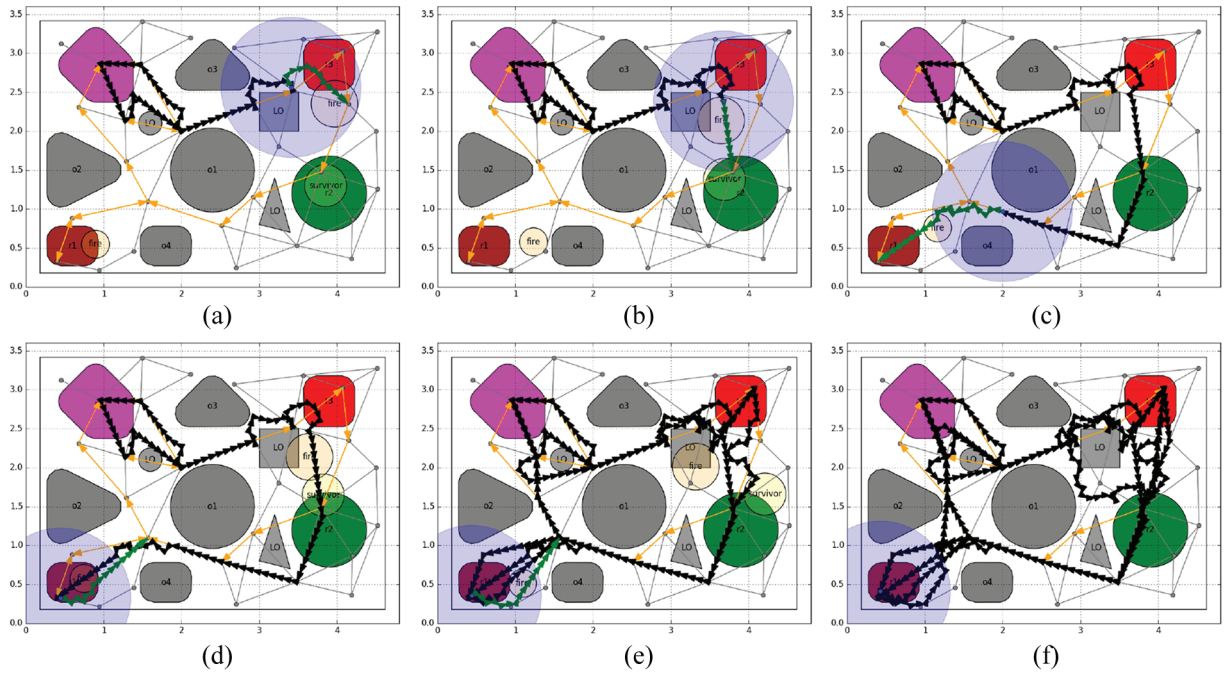


Fig. 7. Consider the environment from Figure 5, and three local a priori unknown obstacles labeled *LO* (light gray). There are also three dynamic requests, two *fire* and a *survivor*. (a) Replanning to service the *fire* request. (b) Replanning needed due to *fire* request's movement. (c) Servicing the second *fire* request. (d) After completing the surveillance prefix. (e) After completing the first surveillance cycle. (f) After completing the second surveillance cycle. The circles around the on-line requests delimit their corresponding service area. The sensing area of the robot is shown as a light blue circle of radius 0.9 around the current position of the robot (blue dot), see Figure 5. The gray lines and dots represent the global transition system $\mathcal{T}_G$, while the satisfying off-line policy is shown as orange arrows. The trajectory of the robot is shown as a sequence of black arrows. Figure (a) shows the trajectory of the robot after a few steps, and the on-line algorithm had to replan to service the upper-right *fire* request. Owing to the request's movement, the trajectory is modified to ensure servicing, see (b). Note that the plan visits region $r3$, but does not need to visit the global state contained in $r3$. The local plan is connected to the global state with minimum potential given current progress along the specification, i.e., visit of the colored regions. The last *fire* request is serviced as shown in (c). The trajectory after completing the prefix, the first cycle, and second cycle are shown in (d), (e), and (f), respectively.
Note: Colour version of the figure is available online.

ending at p_{accept}. The robot must visit x_{accept} infinitely many times, and is the starting and ending configuration of a surveillance cycle.

We ran the off-line planner Algorithm 1 100 times and obtained an average execution time of 696 ms (minimum 77, maximum 5575, standard deviation (SD) 799), out of which the average of the incremental search procedure Algorithm 2 was 89 ms (minimum 6, maximum 713, SD 110). The resulting global transition system had a mean size of 37.39 states (minimum 12, maximum 92, SD 16.78) and 239.64 transitions (minimum 22, maximum 1102, SD 198.62), while the corresponding product automaton had a mean size of 142.67 states (minimum 41, maximum 356, SD 65.84) and 909.45 transitions (minimum 78, maximum 4215, SD 757.56). The Büchi automaton corresponding to ϕ_1 in (7) had 5 states and 19 transitions.

In each surveillance cycle, three dynamic requests are created: two *fire* and one *survivor* shown in Figure 7. We ran the on-line path planner Algorithm 6 to complete 100 surveillance cycles. The trajectory generated by the on-line algorithm at several of its iterations is shown in Figure 7. A local transition system $\mathcal{T}_L$ generated to service a locally sensed *fire* request is shown in Figure 8.

During the simulation, the local path planner Algorithm 7 was executed 10,579 times. The overall execution time dedicated to local planning (lines 9–10 of Algorithm 6) for a single surveillance cycle was on average 9.56 ms (minimum 0.0159, maximum 2,162.05, SD 84.046). The mean size of the generated local transition system is 54.56 (maximum 711, SD 87.436). The path planning algorithm computed local paths that serviced 224 on-line requests. Thus, we can conclude that the robot was able to satisfy the local mission specification in almost all cases while also ensuring the satisfaction of the global specification ϕ_1 in (7).

6.1.2. High-dimensional simulations. In this section, we study the performance of the proposed algorithms with respect to the dimension of the configuration space on synthetic cases. A case study involving an arm of a Baxter robot is presented in Section 6.2.2, where the configuration space has dimension six. The simulations within this section were performed on the desktop system described previously.

Consider a fully actuated point robot with the workspace shown in Figure 9, and the unit hypercube as its

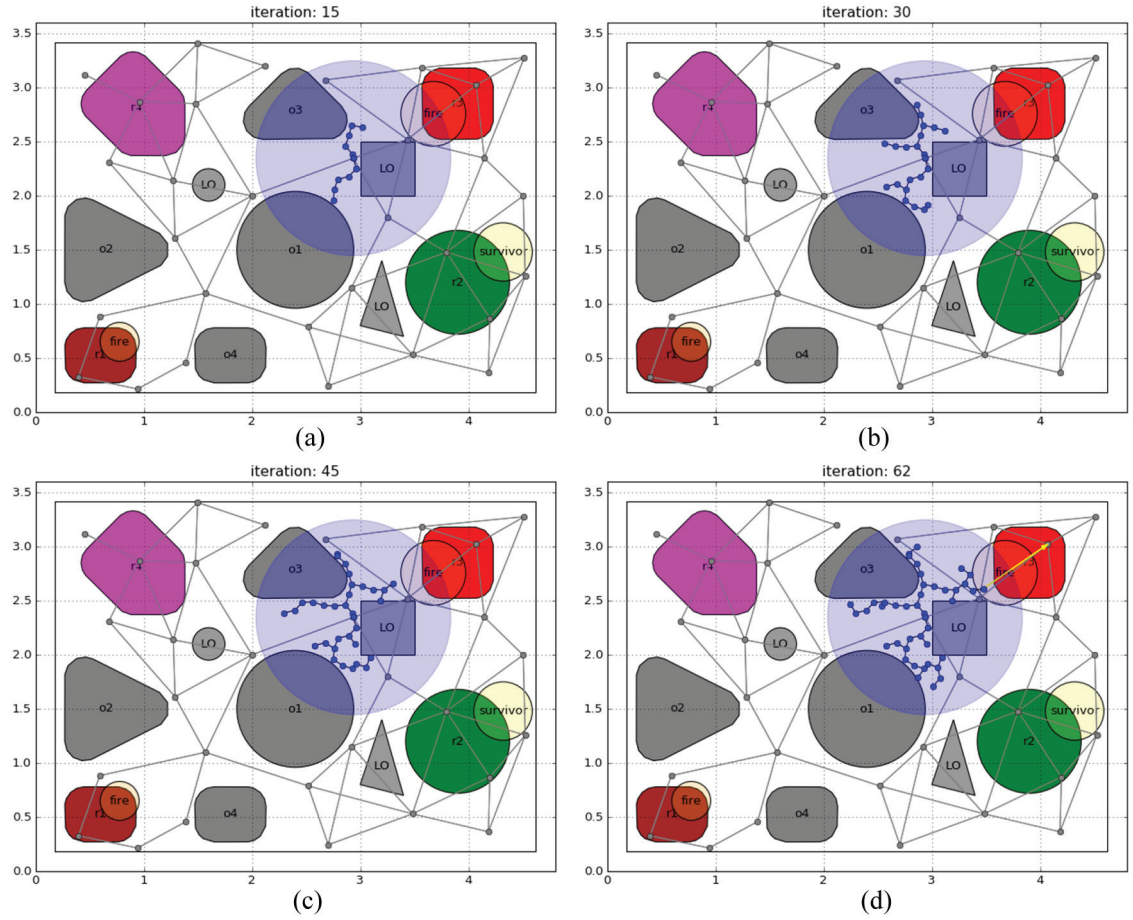


Fig. 8. A local transition system $\mathcal{T}_L$ is generated as an RRT tree within the sensing area of the robot at configuration $x_c = (2.94, 2.35)$. The gray lines and dots represent the global transition system $\mathcal{T}_G$. The yellow arrow represents the connection to the global transition system from the leaf of the tree that satisfies the detected request.

configuration space $\mathcal{C} = [0,1]^n$. The submersion $\mathcal{H}_2(x) = (x_1, x_2)$ projects $x \in \mathcal{C}$ onto the planar workspace spanned by its first two components. The dimension n of $\mathcal{C}$ is varied from 3 to 19.

Within the configuration space we have the following regions: $r1 = [0, 0.2] \times [0, 0.2] \times [0,1]^{n-2}$, $r2 = [0.25, 0.4] \times [0.4, 0.55] \times [0,1]^{n-2}$, $r3 = [0.7, 1] \times [0.4, 0.6] \times [0,1]^{n-2}$, $r4 = [0, 0.5] \times [0.9, 1] \times [0,1]^{n-2}$, $o1 = [0.2, 0.3] \times [0.3, 0.35] \times [0,1]^{n-2}$, $o2 = [0.15, 0.2] \times [0.4, 0.6] \times [0,1]^{n-2}$, and $o3 = [0.5, 0.55] \times [0.3, 0.8] \times [0,1]^{n-2}$. These configuration space regions are obtained from the regions of interest and obstacles in the workspace (Figure 9) as pre-images of $\mathcal{H}_2$. General methods to compute configuration space regions from workspace ones exist for wide classes of robots (Latombe, 2012; LaValle, 2006; Lozano-Perez, 1983).

In all simulations, the specification is to "visit regions $r1$, $r2$, $r3$, $r4$ infinitely many times, while always avoiding regions $o1$, $o2$, $o3$." The LTL formula corresponding to this specification is

$$\phi_2 = (\Diamond r1 \wedge \Diamond r2 \wedge \Diamond r3 \wedge \Diamond r4 \wedge \neg(o1 \vee o2 \vee o3)) \quad (8)$$

The corresponding Büchi automaton has 5 states and 19 transitions. The initial configuration is $x_0 = (0.1, \ldots, 0.1)$.

Similar to the planar case, we executed the off-line planner Algorithm 1 100 times for each configuration space with dimension $n \in \{3, \ldots, 19\}$. The results are shown in Figure 10. Both the number of iterations Figure 10(a) and execution time Figure 10(b) grow exponentially with the dimension of the configuration space. However, the sizes of the computed transition systems Figures 10(d) and (e), and corresponding product automata Figures 10(f) and (g) seem to grow much slower.

To test the on-line planner Algorithm 7 we ran it to complete 100 surveillance cycles for $n \in \{3, \ldots, 19\}$. The robot is equipped with a sensor that detects local obstacles and requests in the configuration space. The shape of the sensing volume is a ball of radius $0.25^{\frac{1}{n}}$ around the current robot's configuration. We consider three local obstacles marked with LO: $[0.45, 0.5] \times [0.75, 0.8] \times [0,1]^{n-2}$, $[0.9, 1] \times [0.5, 0.55] \times [0,1]^{n-2}$, and $[0.75, 0.8] \times [0.2, 0.25] \times [0,1]^{n-2}$. The projection of the three local obstacles in the planar workspace is shown in gray and marked with LO in Figure 9. Three requests are also present: two of type

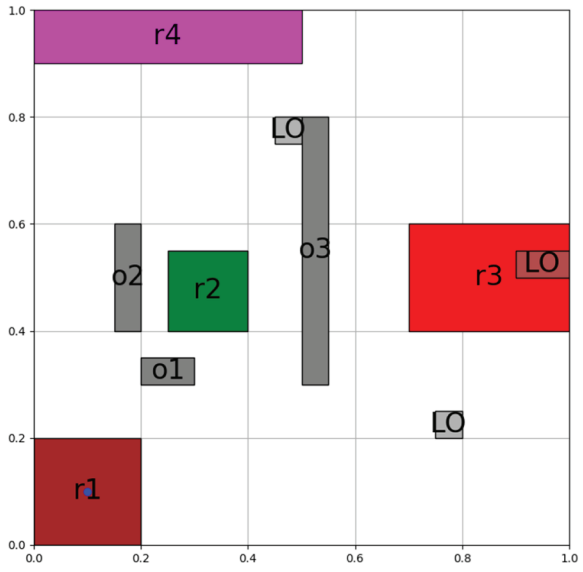


Fig. 9. The planar workspace used in the cases from Section 6.1.2. The environment contains four regions labeled $r1$ (brown), $r2$ (green), $r3$ (red), and $r4$ (magenta) that need to be visited infinitely many times, while avoiding the dark gray obstacles labeled $o1$, $o2$, $o3$. In addition, there are three a priori unknown local obstacles labeled LO (light gray). The initial position of the robot is $y_0 = \mathcal{H}_2(x_0) = (0.1, 0.1)$ and shown as a blue dot.
Note: Colour version of the figure is available online.

1, and one of type 2, and move at constant speed along triangular paths. The three requests' paths are defined by 3-tuples of configurations $([0.25, 0.1, 0.5, \ldots, 0.5]$, $[0.65, 0.1, 0, \ldots, 0]$, $[0.5, 0.25, 0, \ldots, 0])$, $([0.1, 0.6, 0.5, \ldots, 0.5]$, $[0.25, 0.5, 0, \ldots, 0]$, $[0.25, 0.85, 0, \ldots, 0])$, and $([0.5, 0.9, 0.5, \ldots, 0.5]$, $[0.8, 0.9, 0, \ldots, 0]$, $[0.8, 0.6, 0, \ldots, 0])$, respectively. Their servicing radii are set to $0.22\frac{1}{\pi}$, $0.20\frac{1}{\pi}$, and $0.21\frac{1}{\pi}$, respectively. The local specification is to prioritize type 1 requests over type 2 ones.

The results with respect to the number of planner calls, execution time, size of the computed local transition systems, and number of serviced requests are shown in Figures 11(a), (b), (c), and (d), respectively. In the experiments, owing to the random global transition system and the local ones, the number of serviced requests fluctuates between 271 and 303 across dimensions, Figure 11(d), as well as the number of times the local planner is called, Figure 11(a). However, in all cases the on-line planning algorithm took less than 1 s, Figure 11(b), to generate local transitions system with less than 200 states, Figure 11(c).

6.1.3. Comparison with naïve approaches. In this section, we show how the performance of the off-line algorithm is impacted the incremental and sparsity mechanisms are not used. We consider the planar scenario presented in Section 6.1.1 for the comparison. All measures are reported as relative differences, i.e., $(b - a)/a$, where a and b represent the values obtained with the proposed and naïve (modified) procedures, respectively.

Sparsity We consider Algorithm 1 without sparsity. This is achieved by using the *near* function instead of the *far* function with parameter η_2 at line 7. We ran the modified off-line planner again 100 times and obtained an average execution time of 2122 ms (minimum 101, maximum 23,530, SD 3,701), out of which the average of the incremental search procedure Algorithm 2 was 588 ms (minimum 6, maximum 8,259, SD 1,332). It represents a relative increase of 204.89% (minimum 31.17%, maximum 322.06%, SD 363.20%) in the total planning time, and of 560.67% (minimum 0%, maximum 1,058.35%, SD 1,218.81%) for the incremental update procedure. The resulting global transition system had a mean size of 59.9 states (minimum 15, maximum 252, SD 41.34) and 864.41 transitions (minimum 50, maximum 9,647, SD 1,366.22), while the corresponding product automaton had a mean size of 228.8 states (minimum 57, maximum 980, SD 159.8) and 3,277.2 transitions (minimum 189, maximum 37,064, SD 5,213.53). The relative increase of the number of states is 60.2% (minimum 25%, maximum 173.91%, SD 146.36%), and for transitions is 260.71% (minimum 127.27%, maximum 775.4%, SD 587.85%). The product automaton size changed similarly, the relative increase is 60.37% (minimum 39.02%, maximum 175.28%, SD 142.71%) for states, and 260.35% (minimum 142.30%, maximum 779.34%, SD 588.2%) for transitions.

Incremental We consider Algorithm 1 without the incremental maintenance of the product automaton and associated SCCs procedure given in Algorithm 2. Instead, the product automaton update steps at lines 10 and 19 of Algorithm 1 are removed, transitions are always added (i.e., *added* is always true at lines 11 and 20), and the check for a satisfying solution at line 4 is done by constructing a product automaton from scratch. We ran the modified off-line planner again 100 times and obtained an average execution time of 2,618 ms (minimum 121, maximum 63,953, SD 6,853). It represents a relative increase of 276.14% (minimum 57.14%, maximum 1,047.14%, SD 757.7%) in the total planning time. The resulting global transition system had a mean size of 46.1 states (minimum 17, maximum 94, SD 16.25) and 355.24 transitions (minimum 56, maximum 1257, SD 233.94), while the corresponding product automaton had a mean size of 146.82 states (minimum 50, maximum 310, SD 56.42) and 909.45 transitions (minimum 78, maximum 4215, SD 757.56). The relative increase of the number of states is 23.3% (minimum 41.67%, maximum 2.17%, SD −3.16%), and for transitions is 48.24% (minimum 154.55%, maximum 14.07%, SD 17.78%). The product automaton size changed similarly, the relative increase is 2.91% (minimum 21.95%, maximum −12.92%, SD −14.31%) for states, and −1.16% (minimum 58.97%, maximum −17.01%, SD −16%) for transitions.

Discussion In the case where we remove sparsity, all measures, runtime, and models sizes increase significantly. The larger transition system sizes are due to rejecting fewer sampled states, which has a cascading effect on nearest-

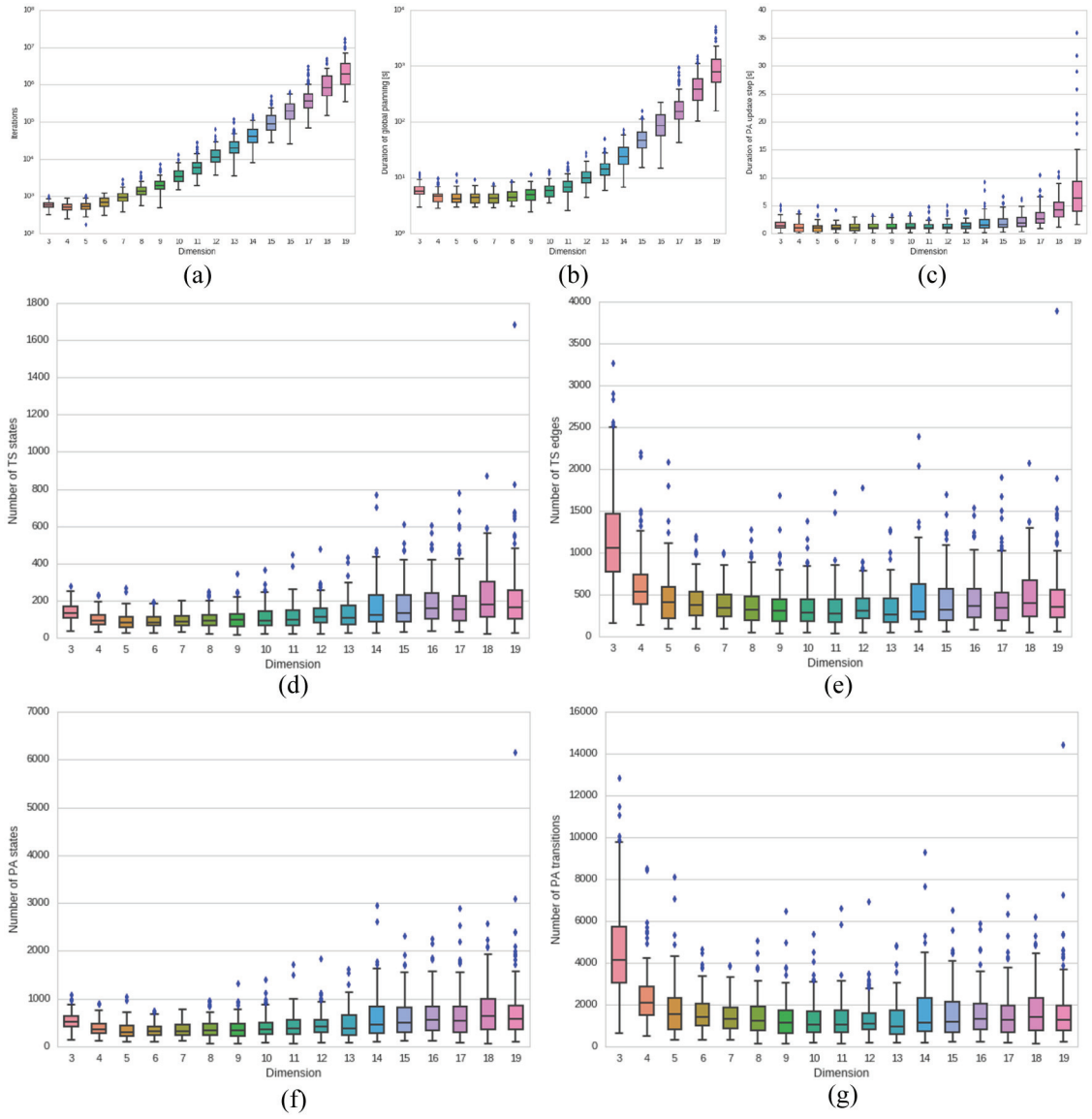


Fig. 10. Boxplots showing the results of across 100 simulated trials for each configuration space of dimension $n \in \{3, \ldots, 19\}$. (a) The number of iterations needed to compute transitions systems containing satisfying paths. The plot highlights the trend of the sample complexity of Algorithm 1 that imposes also the sparsity constraint using the *Far* primitive. The execution time is shown in (b), out of which the duration dedicated to updating the product automaton is shown in (c). The execution times suffer both from the increased number of iterations needed to compute the global transition systems, but also from increased complexity of the primitive functions and collision checks as the dimension of the configuration space increases. However, owing to the sparsity constraint, the sizes of the transition systems and corresponding product automata grow much slower as shown in (d) and (e), and (f) and (g), respectively.

neighbor searches, transition violation checks, product automata, and SCCs updates, and satisfying solutions checks. Even though these operations are incremental, they are more expensive than sampling, especially for more expressive specifications. This also is advantageous for the on-line algorithm. Note that this contrasts approaches of sparsification as a post-processing step as in Dobson and Bekris (2013), where exploration of the free space is more sought, and rejection checks are local. Thus, there is less penalty to creating large transitions systems. Our satisfaction checks are global, and there is a substantial benefit from maintaining sparseness at construction time. The conclusion is that it is better to keep the transition system size small. The non-incremental case also leads to a significant increase in execution time. The main reason for this is the naïve construction of product automata for each satisfaction check. Evidence for this is the small difference in model sizes that can not account for the increase in execution time. We eliminated the transition violation checks and product and SCCs updates that decrease the time spent on each sample. The removal of these operations also leads to adding states and transitions that potentially lead to violation of the specification. It explains the slight increase in transition system sizes. The product automata, on the other

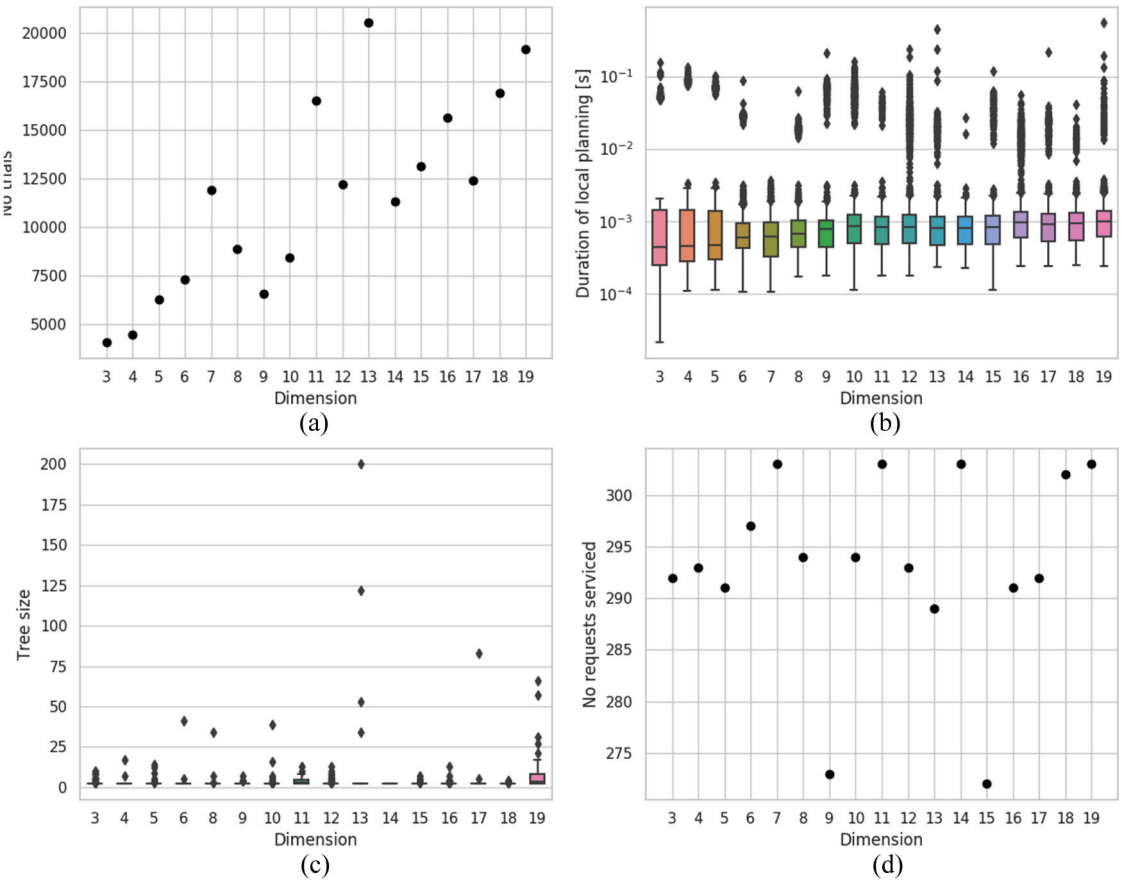


Fig. 11. The figure shows the results of simulated trials with the on-line planner Algorithm 7 over 100 surveillance cycles in configuration spaces of dimension $n \in \{3, \ldots, 19\}$. The number of calls to the procedure is shown in (a), while the boxplots in (b) show the execution time. The sizes of the computed local transition systems, and the number of serviced requests during the runs are shown in (c) and (d), respectively.

hand, do not increase, in size because of the reachability construction from Definition 3, which does not expand violating transitions. The conclusion is that there is a significant benefit from using incremental operations in conjunction with sampling-based approaches.

6.2. Experiments

In this section, we present hardware experiments with two robots, Cozmo and Baxter, in two reactive planning applications.

6.2.1. Reactive planning with Cozmo. Consider the bounded planar environment in Figure 12(a) of size $3.546\,\text{m} \times 2.955\,\text{m}$ containing four colored regions, brown $r1$, green $r2$, red $r3$, and blue $r4$, and black obstacles $o1$, $o2$, and $o3$. The boundary of the environment is marked by the black rim. A Cozmo robot (Figure 12(b)) is deployed to perform a persistent surveillance mission to "visit all colored regions infinitely often while always avoiding all black obstacles" captured by the LTL formula:

$$\square(\diamondsuit r1 \wedge \diamondsuit r2 \wedge \diamondsuit r3 \wedge \diamondsuit r4 \wedge \neg(o1 \vee o2 \vee o3)) \qquad (9)$$

The robot's position is tracked using the OptiTrack indoor localization system, and its initial position is $x_0 = (0.8865, 1.4775)$, where the origin of the reference system is the bottom-left corner.

Three tagged cubes shown in Figure 12(b) are present in the environment corresponding to *a priori* unknown dynamic requests, two *fire* and one *survivor*, that the Cozmo robot needs to service. The priority function *prior* is defined such that *survivor* request have higher priority than *fire* request. LEDs on the cubes mark whether the requests are active (LEDs on), or have been serviced (LEDs off). The cubes are detected using the Cozmo's on board RGB camera, while their position is given by OptiTrack. Note that during the mission, we changed the positions of the cubes, and we reset all cubes to the active state at the beginning of each surveillance cycle.

Algorithm 1 was used to generate a global transition system with 22 states and 72 transitions containing a satisfying policy with a prefix and suffix of lengths 18 and 21, respectively, see Figure 12(c). The Buchi automaton corresponding to (9) had 5 states and 19 transitions, while the induced product automaton had 85 states and 276 transitions. A few iterations of the global off-line algorithm are

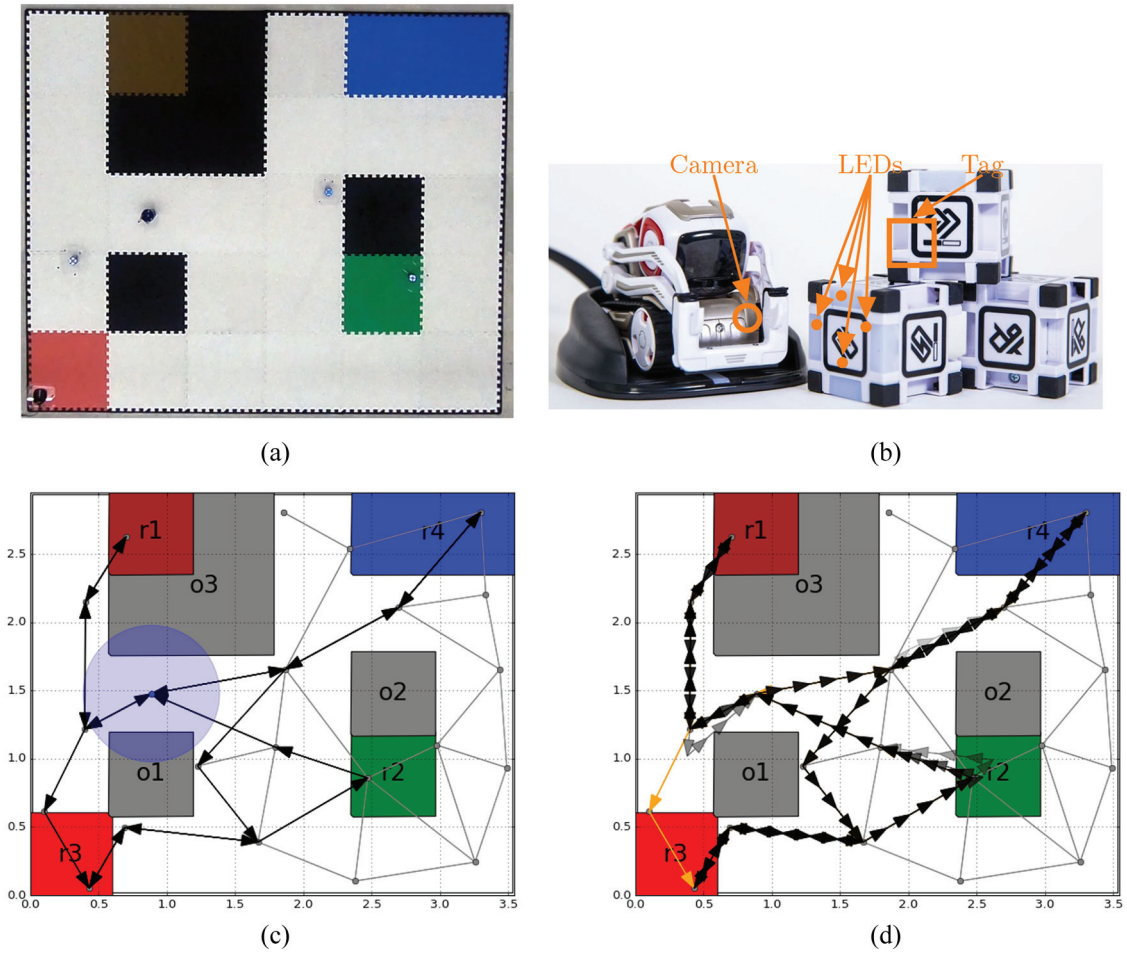


Fig. 12. The experimental area is shown in (a) and contains four colored regions of interest, and three black obstacles (shown in gray for visibility in (c) and (d)). The Cozmo robot, and three tagged and powered cubes are placed in the environment (b). A global transition system is shown in (c) as gray dots and lines. It contains a satisfying path shown as black arrows. The trajectory of the Cozmo robot obtained by executing the on-line Algorithm 6 is shown in (d) as black arrows, where the off-line policy is shown in orange.

shown in Figure 13. The procedure took 231 iterations and 496 ms to compute the global TS. The computation of the potential function using Algorithm 3 took 2.71 ms.

The on-line Algorithm 6 was used to execute the prefix and two surveillance cycles with the Cozmo robot. The algorithm was executed 251 times, and serviced 5 on-line requests. The RRT trees had an average size of 6 (minimum 2, maximum 8, SD 2.44) and took on average 3 ms (minimum 0.983, maximum 43.353, SD 3.78) to compute. The obtained trajectory is shown in Figure 12(d), where we used a simple path following algorithm to drive the Cozmo along the computed local policies. The experiment is shown in Extension 1 (Appendix 1).

6.2.2. Experiments with Baxter's arm. Consider a Baxter robot in an environment containing a table and three areas above it representing pans that the robot needs to persistently attend, see Figure 14(a). We perform planning for the 6-DoF right arm of Baxter with configuration space $\mathcal{C} = [-3., 3.] \times [0., 2.6] \times [-1.69, 1.69] \times [-2.1, 1.] \times [-3., 3.] \times [-1.5, 2.]]$. The workspace $\mathcal{D} \subseteq \mathcal{H}(\mathcal{C})$ is the reachable part of the 3D space around the robot without collisions obtained from the forward kinematics map $\mathcal{H}$. The three cylindrical regions of size $\pi(0.15\text{m})^2 \times 0.27\text{m} = 0.0191\text{m}^3$ associated with the pans are marked with *region*1 (red), *region*2 (green), and *region*3 (blue), respectively. The rectangular volume of size $0.76\text{m} \times 1.22\text{m} \times 0.67\text{m} = 0.6212\text{m}^3$ above the table is labeled with *table*. The global mission specification is to "Attend all pans infinitely may times while staying above the table.", which is translated to the LTL formula

$$\square(\diamondsuit region1 \wedge \diamondsuit region2 \wedge \diamondsuit region3 \wedge table) \qquad (10)$$

The position of the pans, the table, and Baxter's right arm end-effector are obtained from the OptiTrack motion capture system. The initial configuration is $x_0 = [0, 0, 0, 0, 0, 0]$ corresponding to the arm pose shown in Figure 14.

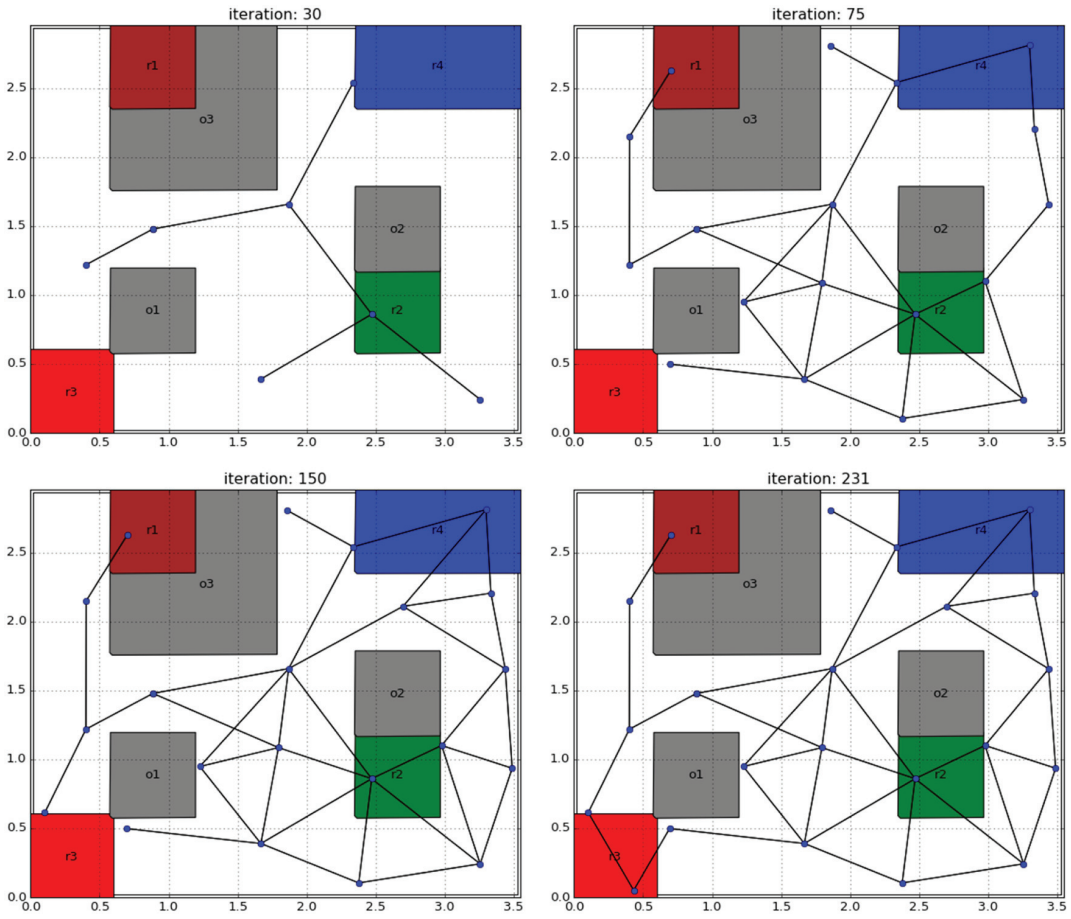
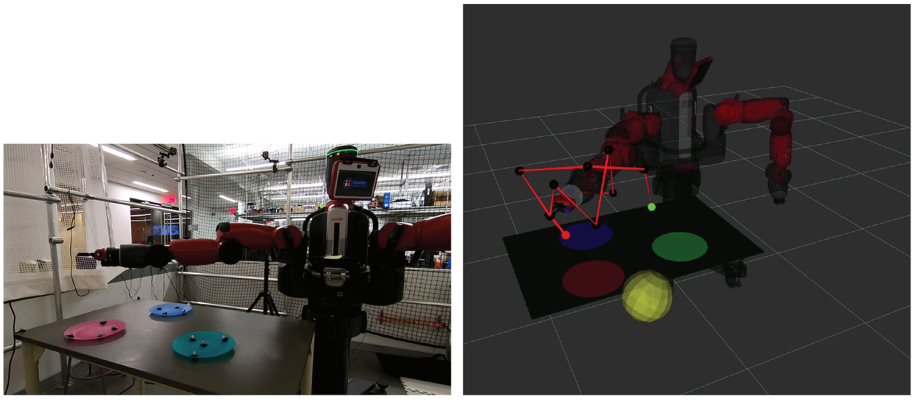


Fig. 13. Transition systems obtained at earlier iterations corresponding to the solution shown in Figure 12(c) (to be read from left to right and top to bottom). The blue dots and black lines represent the states and transitions of $\mathcal{T}_G$, respectively.
Note: Colour version of the figure is available online.



(a) (b)

Fig. 14. The workspace of the robot is shown in (a), where three cylindrical regions corresponding to pans are present atop a table. Baxter's right arm is in its initial configuration. The submersion of a satisfying off-line policy into the workspace is shown in (b) as black balls connected by red ribbons. The policy drives Baxter's end-effector to the three regions while always staying above the table.
Note: Colour version of the figure is available online.

A local request type is defined with label *reactive_region*, and service radius 0.3m. Servicing is performed in the workspace, and the end-effector needs to be within the service radius to satisfy the request. A local request is visible to the robot if it is a 0.6m $\times$ 0.6m $\times$ 0.4m rectangular region on centered on top of the table.

The positions in the 3D workspace of the table, global regions, local requests, and end-effector are given by the OptiTrack in-door localization system, see Figure 14(a). The reference frame is fixed at Baxter's waist with the x axis pointing away from its front, and the z-axis pointing vertically upward. Forward kinematics, simulations, and execution are done using MoveIt (Sucan and Chitta, 2018).

The global transition system was generated using Algorithm 1 and had 32 states and 66 transitions. The computed satisfying policy had a prefix and suffix of lengths 20 and 21, respectively. The workspace path associated with the policy is shown in Figure 14(b). The Buchi automaton corresponding to (10) had 4 states and 13 transitions, while the induced product automaton had 94 states and 196 transitions.

The on-line Algorithm 6 was used to execute the prefix and one surveillance cycle with the Baxter robot. The algorithm was executed 96 times, and serviced one on-line requests. The RRT tree computed to service the request had a size of 21, and took on 40s and 50 iterations to compute. The end-effector's trajectory and experiment are shown in Extension 1 (Appendix 1).

7. Conclusions and future work

We presented a sampling-based framework for motion planning with long-term temporal logic goals while also satisfying short-term reactive requirements. The proposed approach is decomposed into (a) an off-line planner that generates a global transition system containing satisfying paths with respect to the long-term mission goal expressed as LTL formulae, and (b) an on-line planner that deviated from the global structure to service locally sensed requests, while always maintaining satisfaction with respect to the global mission. Both planners are based on sampling-based algorithms, RRG for the off-line planner to capture infinite satisfying paths, and RRT for the local deviations. They also leverage recent algorithms and results in incremental computing, LTL monitoring, and potential functions for Buchi product automata. We show that the algorithms are probabilistically complete, the generated global transition systems are sparse graphs, and the off-line algorithm has the best possible complexity (under mild assumptions). Finally, we highlighted the features of the algorithms and their performance in simulated and experimental trials involving ground robots and manipulators.

The present work has already been extended in Vasile et al. (2016) to stochastic robots, in Penedo Álvarez et al. (2016) and Vasile et al. (2017a) to timed specifications using language-guided and robustness approaches, respectively, and in Vasile et al. (2017b) and Karlsson et al. (2018) to the self-driving vehicles domain. Plans for future work include further improvements of the algorithms using language-guided and robustness techniques to bias sampling and increase convergence. Moreover, specializing the algorithms for classes of robots (e.g., manipulators versus robots with complex dynamics such as cars and quadrotors), and associated tasks would allow the use of heuristics to speed up the planners. Environmental uncertainty and timing constraints are also difficult challenges that need further consideration.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was partially supported by the ONR (Grant Nos. MURI N00014-09-1051 and MURI N00014-10-10952) and by the NSF (Grant Nos. NSF CNS-1035588 and NSF NRI-1426907).

ORCID iD

Cristian Ioan Vasile https://orcid.org/0000-0002-1132-1462

Supplemental material

Supplemental material for this article is available online.

Note

1. In this article, we will assume that we have access to such a function. For more details about planning under differential constraints, see LaValle (2006).

References

Agha-mohammadi Aa, Chakravorty S and Amato NM (2014) FIRM: Sampling-based feedback motion-planning under motion uncertainty and imperfect measurements. *The International Journal of Robotics Research* 33(2): 268–304.

Aguiar AP and Hespanha JP (2007) Trajectory-tracking and path-following of underactuated autonomous vehicles with parametric modeling uncertainty. *IEEE Transactions on Automatic Control* 52(8): 1362–1379.

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

Bauer A, Leucker M and Schallhart C (2007) Runtime verification for LTL and TLTL. Technical Report TUM-I0724, Institut für Informatik, Technische Universität München.

Baykal C, Bowen C and Alterovitz R (2019) Asymptotically optimal kinematic design of robots using motion planning. *Autonomous Robots* 43(2): 345–357.

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

Belta C, Yordanov B and Gol EA (2017) *Formal Methods for Discrete-Time Dynamical Systems*. Berlin: Springer.

Bhatia A, Kavraki LE and Vardi MY (2010) Sampling-based motion planning with temporal goals. In: *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2689–2696.

Burns B and Brock O (2007) Sampling-based motion planning with sensing uncertainty. In: *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3313–3318.

Chen Y, Tumova J and Belta C (2012) LTL robot motion control based on automata learning of environmental dynamics. In: *IEEE International Conference on Robotics and Automation (ICRA)*, Saint Paul, MN, pp. 5177–5182.

Choset H, Lynch KM, Hutchinson S, et al. (2005) *Principles of Robot Motion: Theory, Algorithms, and Implementations*. Boston, MA: MIT Press.

Conway JH and Sloane NJ (1999) *Sphere Packings, Lattices and Groups*. 3rd edn. New York: Springer-Verlag.

Cranen S, Groote JF and Reniers M (2010) A linear translation from LTL to the first-order modal μ-calculus. Technical Report 10-09, Computer Science Reports, Eindhoven University of Technology.

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

Ding XC, Lazar M and Belta C (2014) LTL receding horizon control for finite deterministic systems. *Automatica* 50(2): 399–408.

Dobson A and Bekris KE (2013) Improving sparse roadmap spanners. In: *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4106–4111.

Dobson A, Solovey K, Shome R, Halperin D and Bekris KE (2017) Scalable asymptotically-optimal multi-robot motion planning. In: *International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, pp. 120–127.

Duret-Lutz A, Lewkowicz A, Fauchille A, Michaud T, Renault E and Xu L (2016) Spot 2.0 – a framework for LTL and ω-automata manipulation. In: Artho C, Legay A and Peled D (eds.) *International Symposium on Automated Technology for Verification and Analysis*. Cham: Springer, pp. 122–129.

Frazzoli E, Dahleh MA and Feron E (2000) Trajectory tracking control design for autonomous helicopters using a backstepping algorithm. In: *American Control Conference (ACC)*, Vol. 6, pp. 4102–4107.

Gastin P and Oddoux D (2001) Fast LTL to Büchi Automata Translation. In: Berry G, Comon H and Finkel A (eds.) *13th International Conference on Computer Aided Verification (CAV)* (Lecture Notes in Computer Science, Vol. 2102). Paris, France: Springer, pp. 53–65.

Haeupler B, Kavitha T, Mathew R, Sen S and Tarjan RE (2012) Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Transactions on Algorithms* 8(1): 3:1–3:33.

Hagberg AA, Schult DA and Swart PJ (2008) Exploring network structure, dynamics, and function using NetworkX. In: *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Pasadena, CA, pp. 11–15.

Hauser K (2011) Randomized belief-space replanning in partially-observable continuous spaces. In: Hsu D, Isler V, Latombe JC and Lin MC (eds.) *Algorithmic Foundations of Robotics IX*. Berlin: Springer, pp. 193–209.

Hauser K and Zhou Y (2016) Asymptotically optimal planning by feasible kinodynamic planning in a state–cost space. *IEEE Transactions on Robotics* 32(6): 1431–1443.

He K, Lahijanian M, Kavraki LE and Vardi MY (2017) Reactive synthesis for finite tasks under resource constraints. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5326–5332.

Kantaros Y and Zavlanos MM (2019) Sampling-based optimal control synthesis for multi-robot systems under global temporal tasks. *IEEE Transactions on Automatic Control* 64(5): 1916–1931.

Karaman S and Frazzoli E (2009) Sampling-based motion planning with deterministic μ-calculus specifications. In: *IEEE Conference on Decision and Control (CDC)*, Shanghai, China, pp. 2222–2229.

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

Karaman S and Frazzoli E (2012) Sampling-based optimal motion planning with deterministic μ-calculus specifications. In: *American Control Conference (ACC)*, pp. 735–742.

Karlsson J, Vasile CI, Tumova J, Karaman S and Rus D (2018) Multi-vehicle motion planning for social optimal mobility-on-demand. In: *IEEE International Conference on Robotics and Automation (ICRA)*, Brisbane, Australia, pp. 7298–7305.

Kavraki LE, Svestka P, Latombe JC and Overmars MH (1996) Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation* 12(4): 566–580.

Kingston Z, Moll M and Kavraki LE (2018) Sampling-based methods for motion planning with constraints. *Annual Review of Control, Robotics, and Autonomous Systems* 1(1): 159–185.

Kleinbort M, Solovey K, Littlefield Z, Bekris KE and Halperin D (2019) Probabilistic completeness of RRT for geometric and kinodynamic planning with forward propagation. *IEEE Robotics and Automation Letters* 4(2): x–xvi.

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

Kress-Gazit H, Fainekos GE and Pappas GJ (2007) Where's Waldo? Sensor-based temporal logic motion planning. In: *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3116–3121.

Kress-Gazit H, Lahijanian M and Raman V (2018) Synthesis for robots: Guarantees and feedback for robot behavior. *Annual Review of Control, Robotics, and Autonomous Systems* 1(1): 211–236.

Kuwata Y, Teo J, Fiore G, Karaman S, Frazzoli E and How JP (2009) Real-time motion planning with applications to autonomous urban driving. *IEEE Transactions on Control Systems Technology* 17(5): 1105–1118.

Lahijanian M, Maly MR, Fried D, Kavraki LE, Kress-Gazit H and Vardi MY (2016) Iterative temporal planning in uncertain environments with partial satisfaction guarantees. *IEEE Transactions on Robotics* 32(3): 538–599.

Latombe JC (2012) *Robot Motion Planning*, Vol. 124. Berlin: Springer.

LaValle SM (2006) *Planning Algorithms*. Cambridge: Cambridge University Press.

LaValle SM and Kuffner JJJ (2001) Randomized kinodynamic planning. *The International Journal of Robotics Research* 20(5): 378–400.

Lindemann SR and LaValle SM (2009) Simple and efficient algorithms for computing smooth, collision-free feedback laws over given cell decompositions. *The International Journal of Robotics Research* 28(5): 600–621.

Livingston SC and Murray RM (2013) Just-in-time synthesis for motion planning with temporal logic. In: *International Conference on Robotics and Automation (ICRA)*, pp. 5048–5053.

Livingston SC, Prabhakar P, Jose AB and Murray RM (2013) Patching task-level robot controllers based on a local μ-calculus formula. In: *International Conference on Robotics and Automation (ICRA)*, pp. 4588–4595.

Lozano-Perez T (1983) Spatial planning: A configuration space approach. *IEEE Transactions on Computers* 32(2): 108–120.

Moore J, Cory R and Tedrake R (2014) Robust post-stall perching with a simple fixed-wing glider using LQR-trees. *Bioinspiration and Biomimetics* 9(2): 025013.

Muhayyuddin Moll M, Kavraki LE and Rosell J (2018) Randomized physics-based motion planning for grasping in cluttered and uncertain environments. *IEEE Robotics and Automation Letters* 3(2): 712–719.

Murray RM, Sastry SS and Zexiang L (1994) *A Mathematical Introduction to Robotic Manipulation*. 1st ed. Boca Raton, FL: CRC Press.

Nilsson P, Hussien O, Balkan A, et al. (2016) Correct-by-construction adaptive cruise control: Two approaches. *IEEE Transactions on Control Systems Technology* 24(4): 1294–1307.

Penedo Álvarez F, Vasile CI and Belta C (2016) Language-guided sampling-based planning using temporal relaxation. In: *Workshop on the Algorithmic Foundations of Robotics (WAFR)*, San Francisco, CA, pp. 1–16. Available at: http://wafr2016.berkeley.edu/papers/WAFR_2016_paper_22.pdf

Pierson A, Ataei A, Paschalidis IC and Schwager M (2016) Cooperative multi-quadrotor pursuit of an evader in an environment with no-fly zones. In: *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 320–326.

Raman V, Lignos C, Finucane C, Lee KCT, Marcus M and Kress-Gazit H (2013) Sorry Dave, I'm Afraid I Can't Do That: Explaining unachievable robot tasks using natural language. In: *Robotics: Science and Systems (RSS)*, Berlin, Germany, pp. 1–8.

Reyes Castro LI, Chaudhari P, Tumova J, Karaman S, Frazzoli E and Rus D (2013) Incremental sampling-based algorithm for minimum-violation motion planning. In: *IEEE Conference on Decision and Control (CDC)*, pp. 3217–3224.

Schillinger P, Bürger M and Dimarogonas DV (2018) Simultaneous task allocation and planning for temporal logic goals in heterogeneous multi-robot systems. *The International Journal of Robotics Research* 37(7): 818–838.

Serlin Z, Leahy K, Tron R and Belta C (2018a) Distributed sensing subject to temporal logic constraints. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4862–4868.

Serlin Z, Sookraj B, Belta C and Tron R (2018b) Consistent multi-robot object matching via QuickMatch. In: *International Symposium on Experimental Robotics 2018 (ISER)*. Buenos Aires, Argentina: IFRR, pp. 1–10.

Sucan IA and Chitta S (2018) MoveIt!. http://moveit.ros.org

Talata I (1998) Exponential lower bound for the translative kissing number of d-dimensional convex bodies. *Discrete and Computational Geometry* 19: 447–455.

Tumova J, Reyes-Castro LI, Karaman S, Frazzoli E and Rus D (2013) Minimum-violation LTL planning with conflicting specifications. In: *American Control Conference (ACC)*, pp. 200–205.

Ulusoy A, Marrazzo M and Belta C (2013a) Receding horizon control in dynamic environments from temporal logic specifications. In: *Robotics: Science and Systems (RSS)*, Berlin, Germany, pp. 1–8.

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 on Robotics and Automation (ICRA)*, pp. 331–336.

Ulusoy A, Smith SL, Ding XC, Belta C and Rus D (2013c) Optimality and robustness in multi-robot path planning with temporal logic constraints. *The International Journal of Robotics Research* 32(8): 889–911.

van den Berg J, Abbeel P and Goldberg K (2011) LQG-MP: Optimized path planning for robots with motion uncertainty and imperfect state information. *The International Journal of Robotics Research* 30(7): 895–913.

Vasile CI and Belta C (2013) Sampling-based temporal logic path planning. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Tokyo, Japan, pp. 4817–4822.

Vasile CI and Belta C (2014) Reactive sampling-based temporal logic path planning. In: *IEEE International Conference on Robotics and Automation (ICRA)*, Hong Kong, China, pp. 4310–4315.

Vasile CI, Leahy K, Cristofalo E, Jones A, Schwager M and Belta C (2016) Control in belief space with temporal logic specifications. In: *IEEE Conference on Decision and Control (CDC)*, Las Vegas, NV, pp. 7419–7424.

Vasile CI, Raman V and Karaman S (2017a) Sampling-based synthesis of maximally-satisfying controllers for temporal logic specifications. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vancouver, BC, pp. 3840–3847.

Vasile CI, Tumova J, Karaman S, Belta C and Rus D (2017b) Minimum-violation scLTL motion planning for mobility-on-demand. In: *IEEE International Conference on Robotics and Automation (ICRA)*, Singapore, pp. 1481–1488.

Webb DJ and van den Berg J (2013) Kinodynamic RRT*: Asymptotically optimal motion planning for robots with linear dynamics. In: *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5054–5061.

Wongpiromsarn T, Topcu U and Murray RM (2009) Receding horizon temporal logic planning for dynamical systems. In: *IEEE Conference on Decision and Control (CDC)*, pp. 5997–6004.

Wood DR (2004) Bounded degree acyclic decompositions of digraphs. *Journal of Combinatorial Theory, Series B* 90(2): 309–313.

Appendix. Index to multimedia extensions

Archives of IJRR multimedia extensions published prior to 2014 can be found at http://www.ijrr.org, after 2014 all videos are available on the IJRR YouTube channel at http://www.youtube.com/user/ijrrmultimedia

Table of Multimedia Extensions

Extension	Media type	Description
1	Video	Experiments

Extension 1: The video shows the construction of the global and local transition systems, and the deployment of a robot in a simulated planar environment. The video proceeds to show experiments with a Cozmo robot and a Baxter arm in environments with dynamic, locally sensed requests.