Antlab: a Multi-Robot Task Server

IVAN GAVRAN, MPI-SWS
RUPAK MAJUMDAR, MPI-SWS
INDRANIL SAHA, IIT Kanpur

We present Antlab, an end-to-end system that takes streams of user task requests and executes them using collections of robots. In Antlab, each request is specified declaratively in linear temporal logic extended with quantifiers over robots. The user does not program robots individually, nor know how many robots are available at any time or the precise state of the robots. The Antlab runtime system manages the set of robots, schedules robots to perform tasks, automatically synthesizes robot motion plans from the task specification, and manages the co-ordinated execution of the plan.

We provide a constraint-based formulation for simultaneous task assignment and plan generation for multiple robots working together to satisfy a task specification. In order to scalably handle multiple concurrent tasks, we take a separation of concerns view to plan generation. First, we solve each planning problem in isolation, with an "ideal world" hypothesis that says there are no unspecified dynamic obstacles or adversarial environment actions. Second, to deal with imprecisions of the real world, we implement the plans in receding horizon fashion on top of a standard robot navigation stack. The motion planner dynamically detects environment actions or dynamic obstacles from the environment or from other robots and locally corrects the ideal planned path. It triggers a re-planning step dynamically if the current path deviates from the planned path or if planner assumptions are violated.

We have implemented Antlab as a C++ and Python library on top of robots running on ROS, using SMT-based and AI planning-based implementations for task and path planning. We evaluated Antlab both in simulation as well as on a set of TurtleBot robots. We demonstrate that it can provide a scalable and robust infrastructure for declarative multi-robot programming.

1 INTRODUCTION

Autonomous *cyber-physical systems* are an emerging class of systems with tremendous potential to transform our lives. These are distributed systems which incorporate, in addition to computation and storage, large collections of sensors and actuators, which can be used to interact with and control the physical world autonomously. For example, cyber-physical systems in warehouse management consist of collections of autonomous robots, each of which implements a sophisticated software stack to sense and actuate the physical world and to communicate with other robots or backend servers. The co-ordinated activity of all the robots and the servers together accomplish a higher-level goal, such as large scale logistic management or automated manufacuring pipelines. Such systems are already in production, and trends suggest that many more complex systems are to come [3, 21, 37, 44].

While these systems hold enormous promise, there is little systematic support to develop robotic systems with multiple robots. The current state of the art provides some systems support for programming individual robots communicating with a central server, for example, through infrastructure projects such as ROS [32]. The application-specific code for the robot is written in a

This research was sponsored in part by the ERC Synergy Award "ImPACT" and the DAAD scholarship "Research Stays for University Academics and Scientists."
This article was presented in the International Conference EMSOFT 2017 and appears as part of the ESWEEK-TECS special issue.
2017. 1539-9087/2017/3-ART39 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

traditional programming language such as C++ or Python, but it can use packages that abstract individual sensors or actuators, and can use the messaging infrastructure provided by ROS to manage concurrent, message-passing components. However, ROS was built having in mind a single robot and does not provide support for multi-robot systems or infrastructure to support serving tasks continuously. As such, developing a multi-robot application requires an enormous amount of effort in understanding and managing the complexities of robotics (dynamics, uncertainties, unstructured environments) and concurrent distributed systems (provisioning robots, messaging, fault tolerance).

In this paper, we describe *Antlab*, a programming model and a runtime system to program multiple mobile robots in a declarative way. Antlab aims to close the semantic gap between the declarative specification of tasks at the programmer level and the low-level details of managing individual distributed mobile robots, scheduling and planning, etc.

We provide an abstract programming model and a declarative *task specification* language based on linear-time temporal logic (LTL). Our abstract model represents the underlying world as an *occupancy map* and provides an abstraction for the set of available robots. The occupancy map is a standard data structure in robotics, and represents a discrete abstraction of a physical space using a set of predicates. In the programming abstraction, the user does not program individual robots or even know how many robots there are; instead, the user knows a set of *action primitives* the robots can perform, and declaratively specifies a desired temporal sequence of actions. The propositions in a task can range over spatial locations ("reach location ℓ") as well as action primitives ("pick up," "drop") and the temporal connectives allow expressing application-level behaviors over time. The quantification over robots allows us to specify a task without referring to individual robots (just as query languages allow expressing the intent without specifying specific servers), but also helps express co-ordinated behaviors ("two robots follow each other"). Specifically, the user does not need to know about current states of the underlying robots; it is the responsibility of our run-time system to figure out which robots to assign to a task, how to schedule and plan the task, and how to ensure the system has high throughput.

In order to implement Antlab, we have to solve some core algorithmic and systems challenges. First, we describe a constraint-based combined task and path planner which produces optimal paths for a group of robots and a collection of temporal logic specifications over the occupancy grid. The planning algorithm can be implemented using an SMT solver (e.g. Z3 [30] as in our implementation) or an AI planner supporting LTL constraints.

In practice, one must consider the dynamic and uncertain nature of the robotic environment; for example, there can be dynamic obstacles from other robots fulfilling other tasks in the system, or sensor noise and actuator imprecision. Unfortunately, most synthesis algorithms from specifications do not take into account the dynamic nature of the environment [13, 22, 35, 36, 39–41] or require a complex and a priori specification of all environment events as assumptions to the synthesis procedure [8, 26, 43]. In our experience, the "ideal world" assumption leads to unexpected crashes at run-time as the abstract view does not match the real world, and the "model everything" view does not scale.

Thus, in Antlab, we implement a separation of concerns. We synthesize a path plan for the robots based on the ideal world assumption ignoring all dynamic obstacles and represent the strategy as waypoints. Then, we implement the strategy on a real robot by using a ROS-based navigation stack [12] that considers dynamic obstacles and a dynamic communication protocol that robots use to resolve possible collisions between them. We track the compatibility between the ideal and the actual path, potentially re-synthesizing a strategy if possible or triggering an error to the higher layers.

Finally, we implement a distributed systems layer between the task management and the robots. This layer provides standard systems primitives such as monitoring the status of robots, provisioning robots for task execution, and tracking failures (which get increasingly frequent with growing numbers of robots).

We have implemented Antlab and we have evaluated it on a group of TurtleBots implementing a warehouse scenario where the system has to respond to a stream of "collection" requests which require the robots to visit certain positions, gather objects, and deposit them at other positions while remaining safe and collision-free. Through our experiments, both on real robots and simulations, we show the potential of an end-to-end system like Antlab to scalably implement distributed robotic systems with many robots without individual reference to robots by the user.

In summary, we describe a programming model and runtime system for programming multi-robot applications by integrating the following components.

(1) We provide a declarative programming model based on linear-time temporal logic for multiple mobile robots serving requests in a workspace.
(2) We provide an algorithm for combined task and path planning for multiple robots on top of dynamic robot motion planners.
(3) We provide a run-time system to support *planning*, *robot management*, and *task execution*. Specifically, the run-time system considers real-world deployment issues such as resource management and provisioning as well as dynamic task failures or robot failures.
(4) We demonstrate, using experiments on a group of TurtleBots, the scalability and performance of our runtime, taking into account deployment issues such as task failures, robot failures, and dynamic conflict detection. In particular, we compare the end-to-end performance of robots for different task categories.

2 THE PROGRAMMING MODEL

While Antlab provides a framework for any robotics application, we illustrate it using an example inspired by a multi-robot warehouse management system scenario [21] that we use as a running example. We consider robots moving in a warehouse floor. Parts of the workspace contain objects of interest. Other parts may be blocked by obstacles such as boxes or walls.

A task in this setting consists of a user requesting a set of objects; the task is fulfilled when a set of robots traverses the workspace to collect the requested set of objects and brings all the objects to a special part of the workspace called the *workstation*. The specific formalism for describing tasks is discussed below; informally, each task requires one or more robots to traverse a path in the workspace and carry out certain actions such that (a) the robots fulfill the request (liveness), (b) the robots remain safe, i.e., do not collide with obstacles such as walls, shelves, or other robots.

The task assignment and planning problem is to assign each task to one or more robots, and to synthesize and execute trajectories for each of these robots, such that together the safety and liveness goals are met.

2.1 The System State

We now provide a formal description of the problem.

Occupancy Grids. Robots move in a 2-dimensional or 3-dimensional physical space. However, the *configuration* of the robot may require specifying more dimensions, for example, to provide their velocity and orientation in the space. Thus, in general, we assume that the configuration of a robot is given as a point in some compact subset of the n-dimensional Euclidean space. We represent this configuration space in a discrete way, using an n-dimensional *occupancy grid* [11]. An occupancy grid partitions the continuous space into discrete blocks using a uniform grid along each dimension,

and annotates each block with valuations for a set of predicates. We assume there is a predicate that tells, for each block b, whether it is *occupied* or *free*. Each block is assigned a unique identifier by providing the co-ordinates of its center in any fixed co-ordinate system on $\mathbb{R}^n$. By suitably scaling the distance, we can assume that the unit of distance is one block of the workspace. Thus, the identifier for the neighbors of a block can be obtained by adding or subtracting one unit to some co-ordinate. Since we are always interested in compact spaces, we can assume that the identifiers range over a finite set of elements. In what follows, we fix an occupancy grid X, and a set of predicates Π_X which annotate the blocks of the grid.

Robots and System Configurations. We assume a system of N mobile robots, where each robot has a unique identifier r from a fixed set *Rid* of identifiers. Each robot moves in an occupancy grid in discrete time. That is, we fix a discrete time unit τ, and model an individual robot as a dynamical system evolving in discrete steps of τ time units. The *state* σ of an individual robot consists of (1) its position in the space, $\sigma.x$ (which determines a unique block in the occupancy grid) and (2) its velocity configuration, $\sigma.v$, which represents current magnitude and direction of the velocity of the robot. We denote the set of all velocity configurations by V and assume it contains a value 0 denoting that the robot is stationary.[1] A system with N robots consists of the occupancy grid together with the state of all N robots, such that a consistency condition holds: for each robot state $(\mathbf{x}, \mathbf{v})$, we have that the corresponding block of the occupancy grid is marked occupied, and no two robot states have the same positions (i.e., each block of the occupancy grid is occupied by at most one robot).

Example 2.1. In the warehouse example, a predicate $occupied(b)$ is true for a block b if the block is currently occupied by an obstacle, the predicate $obj(b)(o)$ is true if object o is currently in block b, the predicate $at(b)(r)$ is true if a robot with identifier r is currently occupying block b. There may be induced constraints on predicates: for example, $at(b)(r) \Rightarrow occupied(b)$ and $at(b)(r) \Rightarrow \neg at(b)(r')$ if $r \neq r'$.

Motion Primitives. Robots traversing a workspace define a dynamic system whose behaviors are represented as a sequence of configurations and transitions from one configuration to the next. Following the AI planning literature [16, 20, 28], we describe motions using a set of of *motion primitives* $\Gamma(r)$ available to the robot r, which denote simple actions that a robot can perform at any time step. A motion primitive is an abstraction of a low-level dynamical controller. For our purposes, the specific details of the control algorithm are not important.

Associated with each motion primitive $\gamma(r) \in \Gamma(r)$ is a *pre-condition* $pre(\gamma(r))$, which is a formula over the states specifying under which conditions a motion can be executed and a cost $cost(\gamma(r))$ (e.g., energy expenditure) to execute the motion primitive. We write $post(c, \gamma(r))$ for the state of a robot after the motion primitive $\gamma(r)$ is applied to a state c satisfying $pre(\gamma(r))$.

We use $intermediate(c, \gamma(r))$ to denote the set of grid blocks through which the robot may traverse when $\gamma(r)$ is applied at state c, including the beginning and end blocks. A *trajectory* is a sequence of states $c_0 c_1 \ldots$ such that for each $i \geq 0$, there is a motion primitive $\gamma(r)(i)$ taking the robot from c_i to c_{i+1}.

Example 2.2. Consider a ground robot with five action primitives: $\{\mathsf{H}, \mathsf{L}, \mathsf{R}, \mathsf{U}, \mathsf{D}\}$, where the primitive H keeps the robot in the same block in the occupancy grid and the primitives L, R, U and D move the robot to the adjacent left, right, upper, and lower blocks respectively. The availability of a motion primitive may depend on the current state of the robot. For example, if the velocity of the

[1] In a lower-level dynamical model of a robot, one would need to specify the exact co-ordinates of the velocity configuration. We will work at the level of more abstract actions and, hence, we do not need to specify the exact representation of the velocity configuration space.

robot in a certain direction is high, the motion primitive to go in the opposite direction may not be available. This is specified by the pre-condition. For example, $pre(\mathsf{L})$ may be $v(r) = 0$, i.e., the robot is at rest. The post-condition, similarly, states the effect of the action. For example, $post(\mathsf{L})$ specifies the robot is at the block to the left and its velocity is again zero. Assuming L moves exactly one step, we have $intermediate(b, \mathsf{L}) = \{b, L(b)\}$, where $L(b)$ is the block immediately to the left of b.

The runtime behavior of the robots in *Rid* is described by a discrete-time transition system $\mathcal{T}$. A state of the system is a map from each $r \in Rid$ to a state of r. Let σ_1 and σ_2 be two state vectors and γ be a vector containing as elements the motion primitives applied to the robots in *Rid* in state σ_1. Let $\gamma(r) \in \Gamma(r)$ be the motion primitive applied to robot $r \in Rid$ in state σ_1. We define a transition $\sigma_1 \xrightarrow{\gamma} \sigma_2$ iff

- $\sigma_1(r) \models pre(\gamma(r))$ and $\sigma_2(r) = post(\sigma_1(r), \gamma(r))$ for all $r \in Rid$.
- the trajectory of r between the states σ_1 and σ_2 does not pass through an occupied block, that is
  $\forall r \in Rid, intermediate(\sigma_1(r), \gamma(r)) \ \cap \{b \mid occupied(b)\} = \emptyset$.
- The robots do not collide with each other while doing an (atomic) move from state σ_1 to state σ_2, that is
  $\forall r_1 \in Rid, \forall r_2 \in Rid \setminus r_1 : intermediate(\sigma_1(r_1), \gamma(r_1)) \ \cap intermediate(\sigma_1(r_2), \gamma(r_2)) = \emptyset$.

(Note that the complexity of collision avoidance grows quadratically with the number of robots. We discuss in section 3 how this constraint is handled dynamically.) A (multi-robot) trajectory for *Rid* is a sequence $\sigma_0\sigma_1 \ldots$, where for each $i \geq 0$ and $r \in Rid$, the projection $\sigma_i(r)\sigma_{i+1}(r) \ldots$ is a trajectory of robot r. That is, a trajectory for the system is the collection of trajectories of all the robots.

We make the simplifying assumption that for all robots in the system, each motion primitive requires the same unit of time for execution. This assumption may not hold for robots with heterogeneous capabilities. We can extend our approach to systems where motion primitives take different units of time at the cost of making the planning algorithms more complex [33].

2.2 Specifying Tasks: Linear Temporal Logic

LTL. We provide task specifications using *linear temporal logic* (LTL). Let *Rid* be a finite set of indices ranging over identifiers for robots and let Π be a set of *predicates*.

Following the AI planning literature, we consider the robot specific predicates to describe *fluents*, which specify predicates about the current state of the robot such as $at(b)(r)$, or *action primitives*, which specify a capability of the robot. For example, an action primitive such as $pick(o)(r)$ specifies an action by which the robot picks up an object o in one step. In the planning literature, action primitives consist of a name as well as a pre-condition (when an action is available) and an effect (how taking the action changes the fluent state).

We now introduce the logic to specify tasks. The formulas of LTL are defined by the grammar:

$$\varphi \ ::= \ p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \, U \, \varphi_2$$

where p ranges over predicates in Π and action primitives (two-state predicates). We define the derived logical operators $\wedge$, $\Diamond$, and $\Box$ as usual: $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\Diamond\varphi \equiv true \, U \varphi$, and $\Box\varphi \equiv \neg\Diamond\neg\varphi$. Intuitively, $\bigcirc\varphi$ states that φ holds at the next time point, $\Diamond\varphi$ states that φ will hold at some point in the future, $\Box\varphi$ states that φ will continue to hold from the current instant, and $\varphi_1 \, U \varphi_2$ states that φ_2 will eventually hold at some future point and, until that point, φ_1 will continue to hold.

Robot Quantifiers. We extend the basic logic by allowing outermost quantifiers over the set of identifiers. Formally, formulas of *quantified* LTL are built by existentially or universally quantifying identifier variables in the unary predicates in the formula:

$$\psi \ ::= \ \varphi \mid \exists r.\psi \mid \forall r.\psi$$

Notice that in a model with a fixed set of robots, the quantifiers can be desugared into disjunctions or conjunctions over the set of robots.

Similarly, in our formulas, we allow syntactic sugar that quantifies over blocks in the occupancy grid. For example, we write $\forall b.at(b)(r) \Rightarrow free(b)$ to state that the robot r is in a free block.

We define free identifiers and closed formulas in the usual way, by recursion on the structure of the formulas. A formula is *closed* if it does not have any free variables. A *task* is a finite set of closed formulas.

Semantics. Formally, the semantics of LTL is defined over an occupancy grid, an infinite sequence π of truth assignments to the predicates in Π, and an infinite trajectory σ for *Rid*. For an infinite sequence π, and $i \geq 0$, we write $\pi(i)$ for the ith element in the sequence, and $\pi[i]$ for the suffix of π starting from the ith element. The semantic rules are standard:

- $\pi, \sigma \models p$ for a predicate p iff $\pi(0) \models p$.
- $\pi, \sigma \models p$ for an action primitive p iff $(\pi(0), \pi(1)) \models p$.
- $\pi, \sigma \models \varphi_1 \vee \varphi_2$ iff $\pi, \sigma \models \varphi_1$ or $\pi, \sigma \models \varphi_2$.
- $\pi, \sigma \models \neg\varphi$ iff $\pi, \sigma \not\models \varphi$.
- $\pi, \sigma \models \bigcirc\varphi$ iff $\pi[1], \sigma[1] \models \varphi$.
- $\pi, \sigma \models \varphi_1 U \varphi_2$ iff there is some $k \geq 0$ such that $\pi[k], \sigma[k] \models \varphi_2$ and for each $0 \leq j < k$, we have $\pi[j], \sigma[j] \models \varphi_1$.
- $\pi, \sigma \models \forall r.\varphi$ iff $\pi, \sigma \models \bigwedge_r \varphi$ and $\pi, \sigma \models \exists r.\varphi$ iff $\pi, \sigma \models \bigvee_r \varphi$.

Example 2.3. An example *task* in our setting consists of transporting a set of objects $\{o_i \mid i \in I\}$ to the workstation. We specify the requirement of collecting the object o_i as the "nested diamond" temporal logic constraint:

$$\exists r. \left(\begin{array}{c} \Diamond(obj(b)(o_i) \wedge pick(o_i)(r)) \wedge \\ \Diamond(workstation \wedge drop(o_i)(r)) \end{array} \right) \tag{1}$$

for each $i \in I$, where *workstation* is a predicate that describes the location of the workstation. The existential quantification over r specifies that *some* robot has to fulfill the property (go to a location, pick up an object, and, subsequently, deposit it at the workstation).

System and Environment Assumptions. Tasks are always fulfilled under certain assumptions on the system and on the environment. *Safety assumptions* specify invariants that must always hold, and are of the form $\square\varphi$, where φ is a Boolean combination of predicates. *Liveness assumptions* specify conditions that are satisfied infinitely often, and are of the form $\square\Diamond\varphi$, where φ is a Boolean combination of predicates.

An example of an environment safety assumption is the presence of a static obstacle:

$$\square(\neg free(b))$$

An example of a system safety assumption is obstacle freedom:

$$\square(at(b)(r) \Rightarrow free(b))$$

which states that the robot always avoids the static obstacles given by the predicates $\neg free(b)$. An example of an environment liveness assumption is that a location becomes free infinitely often:

$$\square\Diamond free(b)$$

For example, the liveness assumption can encode that a door is infinitely often open, or that a blocked cell is eventually free.

The *planning problem* for a robot asks to compute a trajectory that satisfies the LTL specification

$$\bigwedge_j \Box\psi_j^s \wedge \bigwedge_j \Box\Diamond\psi_j^l \Rightarrow \bigwedge_j \varphi_j^s \wedge \varphi$$

where the antecedent encodes environment safety and liveness assumptions and the consequent encodes the conjunction of system safety specifications and the task specification.

Dynamic Obstacles. A further safety assumption on the system could be collision freedom, that is, no two robots are in the same location at the same time:

$$\forall r.\forall r'.r \neq r' \Rightarrow \Box(\bigwedge_b at(b)(r) \Rightarrow \neg at(b)(r'))$$

or more generally, freedom from colliding with dynamic obstacles. While in principle, all dynamic obstacles can be modeled as part of the workspace and environment assumptions, doing this either makes conservative approximations on the moving obstacles, making the planning problem infeasible for most cases, or makes the planning problem computationally intractable. For example, in the multi-robot scenario, collision avoidance would require global knowledge of the objectives and strategies of all other robots in the workspace.

Instead, Antlab makes a design decision: planning is performed under the assumption that there are *no* dynamic obstacles, but local motion planning and collision avoidance protocols are implemented to "patch" the global plan locally based on local sensing and communication by the robot. In particular, instead of generating *reactive* strategies for the LTL objectives, we implement a plan in two phases. At the high level, we plan under an optimistic assumption (no dynamic obstacles), and generate a sequence of waypoints. At the low level, we implement a local motion planner to find paths between waypoints. The local motion planner avoids obstacles locally and triggers a re-planning step at the high level if the assumptions made at the high level are invalidated and cannot be locally patched (e.g., by a local collision avoidance protocol).

3 ANTLAB IMPLEMENTATION

We now describe our implementation of Antlab. The lowest layer of Antlab, at the level of robots, uses ROS [32]. On top of ROS, we build an actor framework in Python for distributed messaging, based on the XUDD actor framework.[2] The system state is maintained in a database and the robot manager provides an interface to the database.

Figure 1 is presenting the overall architecture of the system. A front-end application server accepts a stream of LTL tasks, possibly initiated concurrently by different users of the system. The back-end consists of two main components: the *System State* and the *Runtime System*. The system state component is a database which maintains the current state of the world (using an *occupancy grid* data structure to represent a map of the world) as well as the current state of all robots.

The Service manager component constitutes the algorithmic core of the runtime system and has two main components - *Robot Manager* and *Task & Path planner*.

The robot manager keeps track of the current position, availability, and state of all the robots. The runtime system buffers all incoming task requests and periodically invokes the task and path planner. The task and path planner assigns the buffered tasks to some of the available robots (not currently executing any other task) and provides a high-level mission plan (a trajectory for each of the assigned robots such that all tasks are satisfied, if possible). We describe the algorithmic core of the planner in Section 4.

[2] https://github.com/xudd/xudd

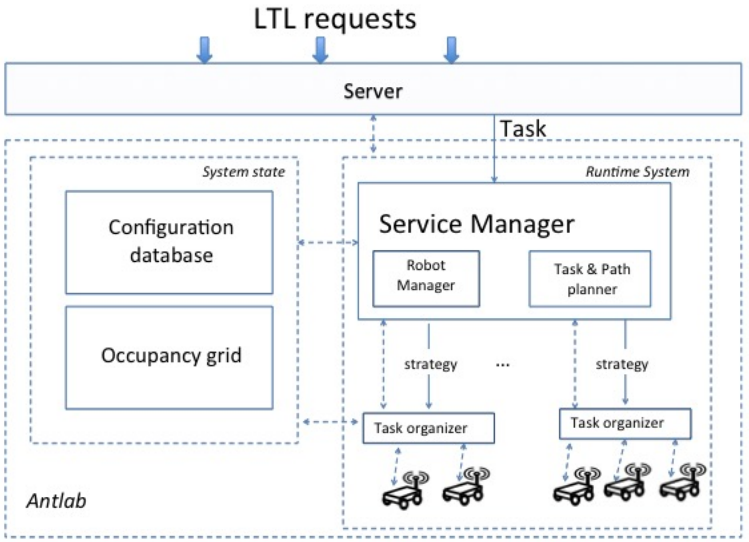


Fig. 1. Antlab system architecture

Once the planning is performed, the runtime system instantiates a task organizer, which monitors the assigned robots and ensures that the tasks are executed as planned. A robot gets the "ideal" plan from the task organizer as a sequence of waypoints (co-ordinates to go to). The robots themselves run a ROS navigation stack to implement the low-level sensing, actuation, and motion planning to move from waypoint to waypoint. This is necessary because the strategy implemented by the task planner may not take into account dynamic obstacles, which can be realized only through dynamic sensing.

The task organizer also communicates with the robots and tracks that the current plan has not failed. A plan can fail if a robot has deviated from the original ideal plan, for example, due to dynamic obstacles, due to sensing or actuation imprecision, or due to violations of the environment assumptions. Finally, a robot itself can fail (e.g., by running out of power) and hence the tasks assigned to it fail as well. The task organizer buffers any failed and unfinished tasks for a future assignment by the task and path planner.

Note that the runtime system is highly concurrent: the server, the system state database, the task organizer, and each individual robot are concurrent components (implemented as actors communicating via message passing). In Figure 1, we denote concurrent messages in the system using dotted arrows.

4 TASK ASSIGNMENT AND PATH PLANNING

Task assignment and execution in Antlab happens at two levels. At the static level, the task and path planner solves a *planning problem*. It takes an occupancy grid, a set of robots, and a set of task specifications, and generates a trajectory for each robot in the occupancy grid such that the combination of trajectories satisfy all the task specification under the assumption that there are no dynamic obstacles (including those induced by other robots in the system).

For example, for a task specified by (1), the output of a successful synthesis problem would consist of robot motion plans which together gather all the objects o_i and bring them to the workstation. The output of the planning problem provides a trajectory per robot as a sequence of motion primitives. Note that we assume the world is determined by the occupancy grid and the environment assumptions and there are no extraneous disturbances.

At the dynamic level, each robot runs a navigation stack to execute its trajectory stepwise. Each robot computes a local trajectory that executes the steps of the trajectory but takes into account dynamic obstacles.

4.1 Problem Definition

We now define the multi-robot task planning problem formally. A *planning problem* instance is given by a five tuple $\mathcal{P} = \langle Rid, I, \Gamma, \Omega, \varphi \rangle$, where Rid is the set of robots, $I : Rid \to X$ maps each robot to a block of the occupancy grid marking its initial location, Γ maps each $r \in Rid$ to a set of motion primitives available to robot r, Ω is a set of environment assumptions, and φ is the LTL specification of the tasks (including all system assumptions).

A multi-robot trajectory is said to be *valid* if it satisfies $\bigwedge \Omega \Rightarrow \bigwedge \varphi$. For example, if φ is $\{\exists r.\Diamond\phi_1(r), \ldots, \exists r.\Diamond\phi_m(r)\}$, a trajectory $\sigma_0\sigma_1 \ldots \sigma_L$ will be valid if for all $\phi \in \{\phi_1, \ldots, \phi_m\}$ there exists a σ_l, $0 \leq l \leq L$, and a robot $r \in Rid$, such that $\sigma_l(r)$ is in a block satisfying ϕ. In general, trajectories are infinite objects. However, we shall only consider finite "lasso-shaped" representations of infinite trajectories given by a prefix and a loop [2]. For each robot $r \in Rid$, for each time instant $t \in \{0, \ldots L\}$, a variable $\gamma(r)(t)$ denotes the motion primitive applied to robot r at step t. A valid trajectory σ is *cost optimal* if $cost(\sigma) \leq cost(\sigma')$ for every valid trajectory σ'.

We define the cost of a trajectory σ as

$$cost(\sigma_0 \ldots, \sigma_L) = \sum_{t=1}^{L} \sum_{r \in Rid} cost(\gamma(r)(t))$$

(it can be viewed as the energy consumption by the robots *in action*). In case the trajectory is infinite, the cost is defined as the weighted sum of the costs of the prefix part and the loop part (with a weight for the loop part significantly higher). Note that the preceding definition does not require the robots to move in sync. Using the "rest" primitive, a robot can wait in its initial state or remain in its final state for an arbitrary amount of time to stretch its length to match with that of the multi-robot system.

We say that an algorithm to solve the planning problem is *sound* if its output, on any problem instance, is a valid trajectory of that instance. We say that the algorithm is *complete* if, for any problem instance, whenever a valid trajectory exists, the algorithm outputs one such valid trajectory.

4.2 Constraint-based Planning

We describe a constraint-based symbolic encoding for cost optimal trajectories using *bounded synthesis* [10]. As mentioned before, we omit adding constraints for collision avoidance and make the optimistic assumption that trajectories of different robots do not collide. We handle collision avoidance dynamically.

Given $\mathcal{P} = \langle Rid, I, \Gamma, \Omega, \varphi \rangle$ and a fixed length L of the trajectory, we model behaviors of the robots as a boolean combination of linear arithmetic constraints. In the system of constraints, the motion primitives of each robot at each state are considered to be the decision variables. For each robot $r \in Rid$, for each time instant $t \in \{0, \ldots L\}$, we consider a variable $\sigma_r(t)$ to track the state of the rth robot and, for each time instant $t \in \{0, \ldots L-1\}$, we consider a variable $\gamma(r)(t)$ encoding the motion primitive applied to robot r at step t. The objective function is to minimize $cost(\sigma)$ over

valid trajectories σ. We add constraints, defined next, to ensure that a valuation to all variables $\sigma_r(t)$ defines a valid trajectory.

Initial state of the trajectory. At the initial state of the trajectory, the robots will be in their initial location and stationary.

$$\forall r \in Rid : \sigma_r(0).\mathbf{x} = I(r) \wedge \sigma_r(0).\mathbf{v} = 0 \tag{2}$$

Conformance between states and motion primitives. For each robot $r \in Rid$, at each time instant t, the state $\sigma_r(t)$ should satisfy the precondition of the motion primitive applied to the robot at time instant t. Moreover, the state $\sigma_r(t)$ should satisfy the postcondition of the motion primitive applied to the robot at time instant $t-1$.

$$\begin{aligned} &\forall r \in Rid,\ \forall t \in \{0,\dots,L-1\} : \sigma_r(t) \models pre(\gamma(r)(t+1)) \\ &\forall r \in Rid,\ \forall t \in \{1,\dots,L\} : \sigma_r(t) = post(\sigma_r(t-1), \gamma(r)(t)) \end{aligned} \tag{3}$$

Safety constraints. The following set of constraints ensures that robots maintain safety constraints when they move from one point to another point.

$$\begin{aligned} &\forall r \in Rid,\ \forall t \in \{0,\dots,L-1\} : \\ &\ \forall b \in intermediate(\sigma_r(t), \gamma(r)(t+1)) : free(b) \end{aligned} \tag{4}$$

We encode environment liveness assumptions $\square\diamond free(b)$ conservatively by specifying the cost of moving to the location b as a cost T incurred by waiting while $\neg free(b)$ holds, before moving from the neighbouring location to b when it becomes free. By setting T much higher in comparison with the costs of all motion primitives, we ensure that a cost optimal trajectory uses the assumption only when no other options are available.

Encoding the LTL specification. Finally, we provide constraints that ensure the trajectory satisfies the formula. We start with an example for an important special case: reaching a set of goal blocks from a set G. The following constraints ensure that the trajectory satisfies such a specification:

$$\forall g \in G, \exists r \in Rid, \exists t \in \{0,\dots,L-1\}.\sigma_r(t).\mathbf{x} \in g \wedge \sigma_r(t).\mathbf{v} = 0. \tag{5}$$

For general LTL specifications, we first replace the quantifiers by disjunctions or conjunctions to generate a pure LTL formula, and generate the constraints capturing the flattened LTL formula using the *eventuality encoding* of [2]. Though a trace that satisfies an LTL formula is given as an infinite execution path of the system, such a trace can be represented by a finite path in two ways: (i) the finite path is a valid prefix of all its infinite extensions (in case the specification is co-safe), (ii) a portion of the finite path can loop to generate a valid infinite path.

Note that we do not plan non-colliding paths for the robots; the rationale is that the planning cost is high, but the gains are poor as the uncertainty of the real-world often causes imperfect executions of plans anyway. Potential collisions are handled locally.

The planning procedure searches over an interval $[L_{\min}, L_{\max}]$ of lengths and generates and solves the constraints for the current choice of $L \in [L_{\min}, L_{\max}]$. If a solution to the set of constraints exists, the trajectories for the robots can be extracted from the solution. If no solution exists, we repeat with a larger value of L. It is important to emphasize that in case of primitives having different cost, an additional optimization process is needed in order to get an optimal solution. (The optimality is otherwise guaranteed by the fact that the solution is found using minimal number of steps and by elimination of unnecessary moves of robots that don't participate in fulfilling the specification). The optimization adds additional constraint on final cost to the formula and iterates until it finds the minimal cost for which the formula is still satisfiable.

While a trajectory generated by the algorithm may not be collision free, we do get the following completeness guarantee [35].

Theorem 4.1. ***Completeness of Multi-Robot Motion Planning.*** *Given an input motion planning problem $\mathcal{P}$ with motion primitives Γ and a trajectory length L, if the system of constraints is not satisfiable, there does not exist a valid trajectory of length L that can be synthesized using the primitives in Γ.*

Solving Constraints. We have implemented two solvers: a symbolic approach based on the SMT solver Z3 [30] and an enumerative search approach based on PDDL planners. We provide an evaluation of these two methods in Section 5. The choice of the solver is transparent to the rest of the implementation.

4.3 Dynamic Implementation of Trajectories

The trajectories of the robots found by the planner are implemented on the individual robots using a motion planner using ROS's navigation stack. The motion planner handles dynamic obstacles as well as potential collisions with other robots. Upon sensing an obstacle, the navigation stack adds it to the local cost map and tries to find a local motion plan to avoid the obstacle and continue with the planned trajectory. In cases where local motion planning and obstacle avoidance cannot find a feasible plan, the navigaton stack starts recovery behaviors and restarts a global planning for the current robot.

We handle potential inter-robot collisions with a local collision avoidance protocol. As noted in [23], robots are active agents and treating them as pure dynamic obstacles leads to inefficient trajectories and —in rare cases— collisions. Furthermore, treating robots as passive obstacles often causes robots to approach too close to each other, which leads to long-lasting recoveries of their local planners and oscillations in the trajectories. Thus, in our implementation, we additionally add local communication capability to robots and implement local collision avoidance strategies. By broadcasting its position and velocity to robots in their neighborhood, and by listening to their messages, a robot can determine its local strategy to avoid collisions.

The approach in [23] uses Optimal Reciprocal Collision Avoidance for non-holonomic robots and takes into account the uncertainties in each robot's localization. Its ROS implementation is provided as well [5]. Unfortunately, we were not able to include it in Antlab as, even though it solves the collision avoidance problem nicely, it introduced problems to ROS navigation stack when going around static obstacles .

Instead, we used a naive priority-based collision avoidance procedure. Each robot has a fixed priority, obtains the configuration of nearby robots and stops as soon as it is close to another robot of a higher priority. If the highest priority robot is blocked due to obstacles presented by other robots (a deadlock), it triggers a re-planning. Collision avoidance is thus not symmetric: some robots have to wait for the others until they are allowed to pass.

5 EVALUATION

We evaluate Antlab's behavior on maps of different size and shape and with different number of robots in the system. We test how well Antlab can handle robot (hardware) failure and dynamic collision situations between the agents. Furthermore, we try to identify situations for which multi-robot planning and assignment is meaningful (where the advantage over simple heuristic is significant). Finally, we examine how the increase in the number of concurrent requests influences the overall performance.

In order to examine different properties, we use different arenas (shown in Figure 3). The *arena: two offices* was obtained by mapping two offices in our building with TurtleBots using the ROS SLAM (Simultaneous Localization and Mapping) gmapping package [19]. All the other arenas are artificially created for simulation. We also tested Antlab in the *empty arena*: a rectangular

Small-Size Map				Medium-Size Map				Large-Size Map			
# Robots	average planning time	average optimization time	cost	# Robots	average planning time	average optimization time	cost	# Robots	average planning time	average optimization time	cost
2	4.10	1.92	40.14	2	9.12	171.29	47.72	2	19.89	218.00	85.65
4	0.92	1.75	15.78	4	2.89	164.73	39.16	4	6.90	111.18	44.94
6	0.87	0.87	12.93	6	1.64	6.64	20.72	6	5.57	110.91	42.01
8	0.70	0.86	11.28	8	1.21	1.74	14.55	8	5.56	54.96	34.85

Table 1. Planning and optimization time and trajectory cost for Φ_2 with increasing number of robots and increasing map size

Property	Small-Size Map			Medium-Size Map			Large-Size Map		
	average planning time	average optimization time	cost	average planning time	average optimization time	cost	average planning time	average optimization time	cost
Repetitive Pick and Drop	0.20	0.50	4.50	1.08	1.92	8.44	2.84	71.47	11.35
Selective Action	0.37	0.62	11.05	1.66	7.57	19.38	6.87	165.92	28.15
Pick and Drop Ordered	0.70	0.86	11.29	1.21	1.74	14.55	5.56	54.96	34.85
Regions Coverage	0.42	0.48	4.44	1.03	1.46	5.44	2.60	0.91	13.75
Sensor measurement	0.58	0.62	13.61	2.40	56.48	22.4	6.97	105.95	35.35

Table 2. Planning and optimization time and trajectory cost for LTL properties for 8 robots and increasing map size

space with no obstacles in it (this is justified by a scenario in which robots can go under all the obstacles, as it is the case for warehouse pods). The proof of concept is done with 3 TurtleBots in the environment mapped in *arena: two offices*. The testing is done in the Stage simulation environment [14], which enabled us to test on different arenas and vary the number of robots in the arena, hence varying the coverage of the area by robots.

5.1 LTL Planning Time and Execution Cost

First, we evaluate Antlab's task assignment and planning on a number of LTL specifications on maps of different sizes and with different number of robots in the system. We consider the following LTL specifications in our experiments:

(Φ_1) *Pick and drop repetitively:* Repeatedly pick an object from the location ℓ and drop it to the location ℓ' (provided that the object would be infinitely often placed at ℓ): $\square\diamond put(\ell) \Rightarrow \exists r.\square\diamond(pick(\ell)(r) \wedge \diamond drop(\ell')(r))$

(Φ_2) *Picking and dropping with enforced order:* Pick $p1$ and $p2$, but once picked, drop the object at $d1$ or $d2$ respectively, before picking anything else: $\exists r_1, r_2 : (\diamond p1(r_1) \wedge \square(p1(r_1) \rightarrow ((\neg p2(r_2) \wedge \neg d2(r_2))\mathcal{U} d1(r_1)))) \wedge (\diamond p2(r_2) \wedge \square(p2(r_2) \rightarrow ((\neg p1(r_1) \wedge \neg d1(r_1))\mathcal{U} d2(r_2))))$

(Φ_3) *Selective action and measurement with safety restrictions:* The propositions $a1$ and $a2$ denote the operation of acting on the certain place; $m1$, $m2$ denote the subsequent measurement at a different place; $s1$ and $s2$ denote locations that should be occupied at each moment, for safety reasons: $\exists r_1, r_2, r_3, r_4 : \diamond(((a1(r_1) \wedge \diamond m1(r_1)) \vee (a2(r_2) \wedge \diamond m2(r_2)) \wedge \square(s1(r_3) \wedge s2(r_4)))$

(Φ_4) *Regions coverage:* From some point onwards, ensure that one of the regions denoted by $s1$, $s2$, $s3$, and $s4$ is always covered by a robot and then eventually point $l1$ is visited: $\exists r_1, r_2, r_3, r_4, r_5 : \diamond(\diamond l1(r_1) \wedge \square(s1(r_2) \vee s2(r_3) \vee s3(r_4) \vee s4(r_5)))$

(Φ_5) *Simultaneous sensor measurement:* Measure sensor values at locations $m1$, $m2$, and $m3$ simultaneously, and report the result at one of the report locations $g1, g2, g3$: $\exists r_1, r_2, r_3 : \diamond((m1(r_1) \wedge m2(r_2) \wedge m3(r_3)) \wedge (\diamond g1(r_1) \vee \diamond g2(r_2) \vee \diamond g3(r_3))$

We created 10 different instances of each of those formulas by picking locations randomly. We used the map *artificial floor* with grid sizes 550 (small, 22×25), 864 (medium, 27×32), and 2250 (large, 45×50). Planning times depend on the map size in grid units. Depending on the motion primitives available, these can be of different size. We used a grid unit of 0.4m.

SMT-based Planning. Table 1 shows times for the SMT-based planner on the different maps as the number of robots increase. Planning (resp., optimization) time is the average time (over 10 instances) to get the first plan (resp. optimal plan). Cost is the optimal value to execute the plan. Increasing the coverage of the arena with robots reduces the cost and makes the planning time shorter (first satisfying instance is reached sooner), as each robot's plan is short. Table 2 shows the same data for Φ_1-Φ_5 for 8 robots.

Anytime Optimization. Time to find a feasible plan is often siginificantly shorter than finding the optimal plan for large maps. Once the first satisfying assignment is found, the optimization process iteratively improves the cost, keeping the number of steps fixed, by invoking the planner with the current upper bound on the cost. Figure 2 shows how the plans approach the optimal (one with 3 robots, and the other with 6 robots on the map). As expected, the biggest improvements are achieved at the very beginning of the process. Thus, the planning can be run in "anytime" mode and stopped once a cost budget is met or a time budget is exceeded.

AI Planning. To compare with an AI planner, we used Temporal Fast Downward (TFD) [15], a classical PDDL planner with temporal logic support. Unfortunately, TFD did not finish on any of these examples. Even on a small empty map (17×19), and a simple reachability objective, TFD was two orders of magnitude slower (103s vs 0.8s); the time was mostly spent in constructing the state space in memory.

Most classical AI planners optimize for reachability. Thus, we compare performance of the SMT planner with Metric-FF [24], a state-of-the-art planner, on reachability objectives $\bigwedge_{i=1}^{n} \Diamond p_i$, where predicates p_i are locations. Each experiment fixed one of three different arenas and averaged over 10 different specifications constructed by picking locations randomly. In order to get faster planning for SMT, we relax the optimality requirement and stop the optimization process once it is within 15 step-units (6 meters) of the optimal plan and set a timeout of 240s for a request. Metric-FF does not provide any optimality guarantees. Table 3 shows that Metric-FF outperforms SMT-planner both in planning time and in cost (due to suboptimality tolerance). Both planners time out on two planning tasks.

Our conclusion is that classical AI planners are still better for simple reachability goals but the SMT planner is the best option for complex LTL goals. A combination of SMT with Metric-FF as an underlying "reachability theory" will be interesting future work.

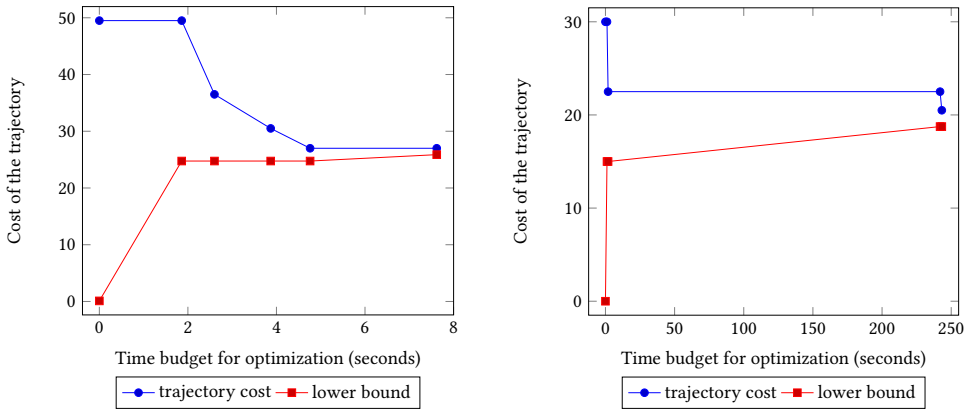
Is Joint Task and Path Planning Necessary? Finally, we evaluate costs and benefits of using the joint task assignment and planning presented in Section 4 in comparison to a naive heuristic which picks the "closest" robot. For general LTL objectives, we do not know how to define closeness, so we fix reachability objectives and pick the robot closest to the goal in Euclidean metric, ignoring obstacles. The heuristic task assignment is "zero cost," but clearly suboptimal as it does not account for obstacles. The task-to-robot assignment of the algorithms presented in Section 4.2 comes at a cost of the time required for path planning. We use the setup to compare against Metric-FF: three arenas, ten reachability specifications. Table 4 shows the summary. For maps for which Euclidean distance is far from the actual travelling distance (such as shoreline or maze), joint planning and task assignment can yield significantly better plans. If Euclidean distance is a good approximation (as in the arena *artificial floor*) then the costs are approximately the same (in this experiment heuristic assignment has a smaller cost due to premature ending of the optimization process in the planner).

Arena name	Avg Planning Time (sec)			Avg Plan Cost		
	shoreline	floor	maze	shoreline	floor	maze
SMT-based	11.56	44.63	20	37	31.37	58
Metric-FF	5.44	6.3	15	22	24.25	54

Table 3. Comparing SMT and AI planning for reachability

Arena name	Planning Time (sec)			Plan Cost		
	shoreline	floor	maze	shoreline	floor	maze
Heuristic assignment	0	0	0	92	25	91
SMT-based assignment and planning	11.56	44.63	20	37	31.37	58

Table 4. Effect of joint task assignment and planning



(a) For a 3 robots instance (b) For a 6 robots instance

Fig. 2. Cost improvement over time

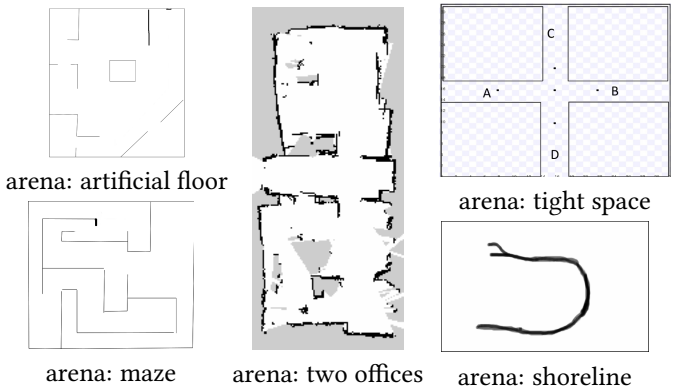


Fig. 3. Workspaces used in the experiments

5.2 Response Time with Concurrency

Next, we measure the effect of concurrent scheduling of tasks –what are the benefits of parallelization and what is the cost of synchronization of robots executing their task at the same time. The experiment is set so that n requests are scheduled concurrently, where each request is a set

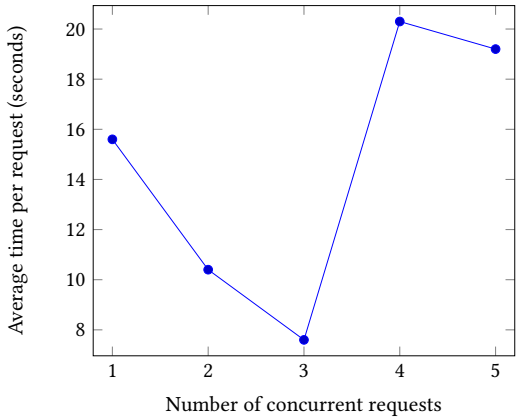


Fig. 4. Execution time vs number of concurrent batches

of tasks, and as soon as a request finishes, the next one is scheduled (thus, there are always n requests executing concurrently). We run 10 requests, each with 3 to 6 tasks, on *artificial floor* with 8 robots and let n grow up to 5. As the results (Figure 4) suggest, response time improves until we reach 4 simultaneous requests. The delays come from two sources: robot synchronization (which is especially significant if two robots get stuck in the deadlock situation) and sub-optimality due to robots being busy with a different batch. That is, if the robots in one part of the arena are busy, and a new request to that part arrives, the planner assigns this request to robots which may be far away. A possible future improvement is to plan based on a partitioning of the arena and only assign close robots even if there is a delay, or to interrupt an active robot to give it a new local task.

5.3 Failure Resilience

In this experiment, we measure the effect of Antlab's fault tolerance and task re-assignment on performance. When a robot crashes, it becomes a static obstacle and the system has to reassign its task. Crashed robots can prevent others from finishing the task, e.g., by blocking a path. We explore how often that might happen.

We model crashes probabilistically: a crash can occur in each second with probability p. The probability of a robot not crashing in k seconds is $(1-p)^k$. We fix the number of tasks to 10, and randomly generate the number of locations per task (the locations are generated so that they are, considering static obstacles, achievable). We run experiments for the coverage of 28 square meters (175 square grid units) per robot and vary the probability p. We measure average time of task completion, as well as the number of locations that Antlab did not manage to reach.

We run 60 trials of each experiment. The parameter p is tested for values $\{0, 0.003, 0.005, 0.006, 0.01\}$ which is equivalent to the following probabilities of staying crash-free for a minute: $\{1, 0.83, 0.74, 0.69, 0.55\}$. The initial setting consists of an obstacle-free arena with 9 robots, and robots are given altogether 24 locations to visit in 10 tasks. For higher values of p, we could not obtain meaningful results because often all the robots would crash before completing the tasks.

Table 5 summarizes the results. The baseline is the execution without crashes, which takes about two and half minutes to complete. The execution time increases with the probability of failure. For $p = 0.01$ the average number of locations which the system was unable to visit is 2.23 but the execution time almost doubles compared to the baseline.

p	Average execution time (sec)	Average number of locations not visited
0	156.36	0
0.003	220.84	0.42
0.005	241.28	1.13
0.006	244.72	1.12
0.01	290	2.23

Table 5. Failure resilience test results

Number of patrolling robots	Execution time with collision avoidance turned on (sec)	Execution time with collision avoidance turned off (sec)
0	365	
1	498	563
2	493	551
3	459	667
4	479	596

Table 6. Navigation in crammed environments

5.4 Tight Space Manoeuvring

Finally, we explore the effect of the dynamic collision avoidance mechanism on performance. Due to its assignment mechanisms, Antlab avoids having more robots then necessary at one place (by assigning multiple tasks to a single robot, minimizing the energy spent). Thus, we construct a specific configuration where many robots are located in the same area and therefore collisions are common. We use *arena: tight space* (Figure 3). Four robots are set to patrolling from side to side ($A \leftrightarrow B$ and $C \leftrightarrow D$). An additional robot that starts in the middle (*task robot*) gets 10 tasks chosen from A, B, C, and D, in a round-robin fashion. It is also forced to visit the central location between each task, causing congestion in the middle. We run this experiment with 0 to 4 other patrolling robots, and set the priority of the task robot as the highest. We run with both collision avoidance turned on and turned off. Note that even with collision avoidance turned off robots will not collide all the time as the lower level navigation stack implements dynamic obstacle avoidance. However, turning collision avoidance on allows robots to communicate their position and goals.

Table 6 shows execution times. We note that without the collision avoidance mechanism turned on, robots run into deadlock situations, where none of their recovery behaviors manages to find the way out and the navigation stack gets stuck. Collision avoidance manages to lower the number of deadlocks, though not to eliminate them completely. The results also show that the execution time does not increase with increasing number of patrolling robots. A possible explanation would be that more robots approaching each other might cause earlier stopping and therefore prevent deadlock-like configurations from happening.

6 RELATED WORK

ROS [32] is a standard software framework for individual robots. Antlab uses ROS at the individual robot level. A continuation project, ROS2 [6], currently in alpha, would handle some of the aspects of Antlab, but does not provide synthesis from higher level declarative specifications. The recently proposed StarL [29] framework unifies programming, specification and verification of distributed robotic systems. None of these systems, however, addresses the specific challenges for managing a robot service, such as managing of robots, dealing with incoming task requests, dynamic obstacle avoidance, and fault-tolerance.

Planning is a classical problem in AI and robotics [4, 16, 20, 27, 34]. As the primary focus of our work is on multiple robots, we focus our discussion on this setting. We compare the performance of AI planners vs SMT for LTL tasks. SMT-based synthesis of motion plans were applied to a single

robot moving in a workspace containing rectangular obstacles [25] and in synthesizing integrated task and motion plans from partially specified tasks [31, 42]. In multi-robot motion planning, an SMT solver was employed to synthesize a plan for a group of robots from a safe-LTL formula [35] and from a specification that requires a group of robots to reach their preassigned goals while avoiding obstacles and collision with other robots [36]. None of these papers deal with a *stream* of incoming tasks and the problem of joint task assignment and planning.

Several prior papers address the problem of generating trajectories for multi-robot systems where the robots are preassigned a set of tasks, whereas Antlab simultaneously assigns robots to tasks and generates trajectories. A subset of these works (e.g. [13, 35, 36, 39]) adopts a centralized approach where a central server is used to synthesize trajectories for a set of robots to reach a set of pre-specified goal locations. The others employ a decentralized prioritized planning (e.g. [22, 40, 41]) where, given a fixed set of tasks, the robots in the system coordinate with each other asynchronously to compute the trajectories. Similarly to our work, in [38] goal assignment and trajectory planning is done simultaneously, but only for simple reachability tasks and requiring that no robot is left idle, rather than optimizing total cost. Drona [9], a framework for distributed mobile robotics, introduces a multi-robot decentralized motion planner, but–unlike Antlab–assumes that robots are preassigned their tasks and that there are no uncertainties in the environment. Antlab can be viewed as a Cyber-Physical Cloud Computing system, as proposed in [7].

Recently, there is an increased interest towards using temporal logic for synthesizing reactive motion plans automatically [8, 26, 43]. However, automated synthesis algorithms scale poorly both with the number of robots and the size of the workspace, and have not proven suitable for multi-robot applications. In order to tackle the scalability issue, several papers synthesize motion plans compositionally (observing a conjunctive LTL formula and synthesizing each conjunct separately) [1, 17], but even though the compositional approach sometimes outperforms centralized ones, the largest examples so synthesized are still far from real world test cases. LTLMoP [18] is a modular toolkit that covers different aspects of creating controllers synthesized from LTL or structured English specifications. The main difference from our work is that LTLMoP still resides in a "clean" world, not taking into account robot crashes, imperfect executions of plans, etc.

7 CONCLUSION AND LIMITATIONS

We have described the design, implementation, and preliminary evaluation of Antlab, an infrastructure for "robots as a service." We consider Antlab as a step towards end-to-end systems for managing and using robot teams, but there are many open directions. The task language of Antlab does not support some typical "swarm" actions, such as coordinated work by many robots ("follow the leader"). It would be interesting to extend the declarative task language to tasks of this nature. Further, LTL is still too low-level to describe complex workflows such as industrial processing.

Scalability in Antlab can be reached only by performing careful tradeoffs when choosing which assignment or planning algorithm to use and how many robots to have in the workspace: the planning procedure is worst case exponential in the size of the workspace and the number of robots. We do not currently support task priorities or plans that pre-empt active robots. As Antlab is built on top of existing infrastructure for single robots (ROS), it is vulnerable to all potential problems at that layer (such as noise and imprecision in sensing and actuation, limited collision-avoidance protocols, etc.). Thus, we cannot give strong end-to-end real-time guarantees about request execution. Providing end-to-end *predictable* multi-robot performance is a difficult problem beyond the scope of this paper.

REFERENCES

[1] R. Alur, S. Moarref, and U. Topcu. Compositional synthesis of reactive controllers for multi-agent systems. In *CAV*, 2016.

[2] A. Biere, K. Heljanko, T. Junttila, T. latvala, and V. Schuppan. Linear encoding of bounded LTL model checking. *LMCS*, 2(5:5):1–64, 2006.

[3] R. Bogue. Growth in e-commerce boosts innovation in the warehouse robot market. *Industrial Robot: An International Journal*, 43(6):583–587, 2016.

[4] H. Choset, K. M. Lynch, S. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. *Principles of Robot Motion.* A Bradford Book, 2005.

[5] D. Claes. collvoid package for ROS. https://github.com/daenny/collvoid.

[6] community project. ROS2. https://github.com/ros2/ros2/wiki. Accessed: November 2016.

[7] S. S. Craciunas, A. Haas, C. M. Kirsch, H. Payer, H. Röck, A. Rottmann, A. Sokolova, R. Trummer, J. Love, and R. Sengupta. Information-acquisition-as-a-service for cyber-physical cloud computing. In *HotCloud*, 2010.

[8] J. A. DeCastro, J. Alonso-Mora, V. Raman, D. Rus, and H. Kress-Gazit. Collision-free reactive mission and motion planning for multi-robot systems. In *ISRR*, 2015.

[9] A. Desai, I. Saha, J. Yang, S. Qadeer, and S. A. Seshia. Drona: A framework for safe distributed mobile robotics. In *ICCPS*, pages 239–248, 2017.

[10] R. Ehlers and B. Finkbeiner. Reactive safety. In *GandALF 2011*, EPTCS 54, pages 178–191, 2011.

[11] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *IEEE Computer*, 22(6):46–57, 1989.

[12] E. M. Eppstein. ROS navigation stack. http://wiki.ros.org/navigation. Indigo version.

[13] M. A. Erdmann and T. Lozano-Pérez. On multiple moving objects. In *ICRA*, 1986.

[14] R. Vaughan et al. Stage simulator. http://wiki.ros.org/stage. Indigo version.

[15] P. Eyerich, R. Mattmüller, and G. Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Towards Service Robots for Everyday Environments*, pages 49–64. Springer, 2012.

[16] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.

[17] E. Filiot, N. Jin, and J.-F. Raskin. Antichains and compositional algorithms for LTL synthesis. *FMSD*, 39(3):261–296, 2011.

[18] C. Finucane, Gangyuan Jing, and H. Kress-Gazit. LTLMoP: experimenting with language, temporal logic and robot control. In *IROS*, pages 1988–1993, 2010.

[19] B. Gerkey and V. Rabaud. Slam gmapping package. https://github.com/ros-perception/slam_gmapping. Indigo version.

[20] M. Ghallab, C. Aeronautiques, C. K. Isi, and D. Wilkins. PDDL: The planning domain definition language. Technical Report CVC TR98003/DCS TR1165, Yale Center for Computational Vision and Control, 1998.

[21] E. Guizzo. Three engineers, hundreds of robots, one warehouse. *IEEE Spectrum*, 45(7):26–34, 2008.

[22] Y. Guo and L.E. Parker. A distributed and optimal motion planning approach for multiple mobile robots. In *ICRA*, 2002.

[23] D. Hennes, D. Claes, W. Meeussen, and K. Tuyls. Multi-robot collision avoidance with localization uncertainty. In *AAMAS*, pages 147–154, 2012.

[24] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

[25] W. N. N. Hung, X. Song, J. Tan, X. Li, J. Zhang, R. Wang, and P. Gao. Motion planning with Satisfiability Modulo Theroes. In *ICRA*, pages 113–118, 2014.

[26] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE Transactions on Robotics*, 2009.

[27] H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R.B. Scherl. GOLOG: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997.

[28] V. Lifschitz and W. Ren. A modular action description language. In *AAAI*, pages 853–859. AAAI Press, 2006.

[29] Y. Lin and S. Mitra. StarL: Towards a unified framework for programming, simulating and verifying distributed robotic systems. In *LCTES*, 2015.

[30] L. De Moura and N. Bjrner. Z3: an efficient smt solver. In *TACAS*, pages 337–340, 2008.

[31] S. Nedunuri, S. Prabhu, M. Moll, S. Chaudhuri, and L. E. Kavraki. SMT-based synthesis of integrated task and motion plans from plan outlines. In *ICRA*, 2014.

[32] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[33] V. Raman, N. Piterman, and H. Kress-Gazit. Provably correct continuous control for high-level robot behaviors with actions of arbitrary execution durations. In *ICRA*, pages 4075–4081, 2013.

[34] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Pearson, 2009.

[35] I. Saha, R. Ramaithitima, V. Kumar, G. J. Pappas, and S. A. Seshia. Automated composition of motion primitives for multi-robot systems from safe ltl specifications. In *IROS*, pages 1525–1532. IEEE, 2014.
[36] I. Saha, R. Ramaithitima, V. Kumar, G. J. Pappas, and S. A. Seshia. Implan: Scalable incremental motion planning for multi-robot systems. In *ICCPS*, 2016.
[37] M. Saska, V. Vonásek, J. Chudoba, J. Thomas, G. Loianno, and V. Kumar. Swarm distribution and deployment for cooperative surveillance by micro-aerial vehicles. *J. Intelligent & Robotic Systems*, pages 1–24, 2016.
[38] M. Turpin, K. Mohta, N. Michael, and V. Kumar. Goal assignment and trajectory planning for large teams of aerial robots. In *RSS*, 2013.
[39] J. van den Berg and M. Overmars. Prioritized motion planning for multiple robots. In *IROS*, 2005.
[40] M. Čáp, P. Novák, M. Selecký, J. Faigl, and J. Vokřínek. Asynchronous decentralized prioritized planning for coordination in multi-robot system. In *IROS*, 2013.
[41] P. Velagapudi, K. Sycara, and P. Scerri. Decentralized prioritized planning in large multirobot teams. In *IROS*, 2010.
[42] Y. Wang, N. T. Dantam, S. Chaudhuri, and L. E. Kavraki. Task and motion policy synthesis as liveness games. In *ICAPS*, page 536, 2016.
[43] T. Wongpiromsarn, U. Topcu, and R. M. Murray. Receding horizon temporal logic planning. *IEEE Trans. Automat. Contr.*, 2012.
[44] M. Wulfraat. Is Kiva systems a good fit for your distribution center? an unbiased distribution consultant evaluation. http://www.mwpvl.com/html/kiva_systems.html. Accessed: October 2016.