Optimal and Dynamic Planning for Markov Decision Processes with Co-Safe LTL Specifications

Bruno Lacerda, David Parker and Nick Hawes

***Abstract*— We present a method to specify tasks and synthesise cost-optimal policies for Markov decision processes using co-safe linear temporal logic. Our approach incorporates a dynamic task handling procedure which allows for the addition of new tasks during execution and provides the ability to re-plan an optimal policy on-the-fly. This new policy minimises the cost to satisfy the conjunction of the current tasks and the new one, taking into account how much of the current tasks has already been executed. We illustrate our approach by applying it to motion planning for a mobile service robot.**

I. Introduction

Markov decision processes (MDPs) are a widely used model for planning under uncertainty, for example to synthesise optimal control policies for robots. In recent years, there has been growing interest in the use of linear temporal logic (LTL) as a task specification language for robot systems, in such domains as optimal control [2], [10], [12], reactive mission planning [4], [13] or multi-robot coordination [7]. LTL is a desirable specification language in the robotics domain for several reasons. First, it provides a powerful and intuitive way of unambiguously specifying a variety of robot tasks. Second, algorithms and tools are available to synthesise correct-by-construction controllers directly from a model of the system and an LTL specification. Thus, LTL provides a means to bridge the gap between a designer's specification for a robot's behaviour and an implementation of a controller that generates behaviour to meet that specification.

Algorithms for controller synthesis from LTL mostly originate from the field of formal verification, where techniques such as model checking and reactive synthesis have received considerable attention. In the context of robotics, where the systems are often modelled stochastically (e.g. using MDPs), techniques from *probabilistic model checking* can be adapted. In this work, we build upon these methods and develop them in an extension of the well-known probabilistic model checking tool PRISM [6].

Probabilistic model checking provides temporal logics to reason not just about the probability that some LTL-specified task is achieved, but also a variety of other quantitative measures using cost (or reward) functions. These might represent, for example, execution time or energy consumption. In this paper, we are interested in generating policies that minimise the expected accumulated cost to achieve one or more tasks. More precisely, these tasks will be ones that can be completed in a finite period of time, and are specified in the *co-safe* fragment of LTL. Tasks are thus not simply about reaching a given target state, but can be *temporally extended* goals that require, for example, a set of states to be visited in a given order. An example of such a task is a mail delivery robot that needs to distribute mail to different rooms in a building, and minimise the time spent in delivery so it can be available to do other tasks as soon as possible.

We show that finding policies which minimise the expected cost to satisfy co-safe LTL formulas in a given MDP model can be reduced to finding policies that minimise the cost to reach a set of states in an MDP composed from the original model and a finite state automaton that represents the LTL formula. We then consider the scenario where multiple LTL-specified tasks are issued dynamically, and show how to re-plan a cost-optimal policy on-the-fly. We implement our methods in the PRISM tool and illustrate their applicability to the mobile robotics domain by using them to perform optimal motion planning for a mobile robot.

Related work. The work in [2], [10] tackles the problem of maximising the probability of satisfying a given LTL formula while minimising the long-term average cost between two states that satisfy a given "optimising" atomic proposition pre-defined by the designer. The main difference to our work is that the aim is to find the policy that reaches the steady-state of the system that minimises an infinite horizon cost function, while our work is more related to transient analysis, where we tackle the problem of finding the policy that minimises the accumulated cost on a finite horizon. This finite horizon minimisation allows us to define our notion of optimal policy in a more straightforward way. In terms of application, the work in [2], [10] is especially suited for the execution of static and persistent tasks that do not have an exact notion of being "completed", e.g., patrolling a building "forever". In our case, however, there is a well-defined notion of a completed task. This enables us to use a more dynamic task allocation procedure, along with on-the-fly replanning, for minimising the mixing of different incoming tasks, defined by the user at different times. The work in [12] also deals with cost optimisation, modelling the system as a weighted transition graph, thus not taking uncertainty into account. In [11], an incremental approach for policy generation that maximises the probability of satisfying a co-safe LTL specification is presented. The incremental element in this approach is related to the addition of new independent agents that have an impact on the ability of the robot to fulfil the specification. Furthermore, the work does not take into account the cost of executing an action.

B. Lacerda, D. Parker and N. Hawes are with the School of Computer Science, University of Birmingham, United Kingdom. {b.lacerda, d.a.parker, n.a.hawes}@cs.bham.ac.uk

II. PRELIMINARIES

A. Notation

We start by clarifying some notation used throughout the paper. Let X be a set. We define: (i) 2^X as the set containing all subsets of X; (ii) X^* as the set containing all the finite sequences of elements of X; and (iii) X^ω as the set containing all the infinite sequences of elements of X. Let $\rho = \rho_0 \dots \rho_n \in X^*$ and $\sigma = \sigma_0 \sigma_1 \dots \in X^\omega$. We define the concatenation of ρ and σ as $\rho \cdot \sigma = \rho_0 \dots \rho_n \sigma_0 \sigma_1 \dots \in X^\omega$.

B. Markov Decision Processes

We will model systems using *Markov decision processes* (MDPs) with atomic propositions labelling the states and costs associated with state-action pairs. An MDP is a tuple $\mathcal{M} = \langle S, \overline{s}, A, \delta_\mathcal{M}, AP, Lab, c \rangle$, where: (i) S is a finite set of states; (ii) $\overline{s} \in S$ is the initial state; (iii) A is a finite set of actions; (iv) $\delta_\mathcal{M} : S \times A \times S \to [0,1]$ is a probabilistic transition function, where $\sum_{s' \in S} \delta_\mathcal{M}(s,a,s') \in \{0,1\}$ for all $s \in S$, $a \in A$; (v) AP is a set of atomic propositions; (vi) $Lab : S \to 2^{AP}$ is a labelling function, such that $p \in Lab(s)$ if and only if the atomic proposition p is true in state s; and (vii) $c : S \times A \to \mathbb{R}_{\geq 0}$ is a cost function, associating each state-action pair with a non-negative value.

To simplify notation, we define the *enabled* actions in s as $A(s) = \{a \in A \mid \delta_\mathcal{M}(s,a,s') > 0 \text{ for some } s' \in S\}$. An MDP model represents the possible evolutions of the state of a system: in each state s, any of the enabled actions $a \in A(s)$ can be selected and the probability of evolving to a successor state s' is then $\delta_\mathcal{M}(s,a,s')$. An infinite *path* through an MDP is a sequence $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots$ where $\delta_\mathcal{M}(s_i, a_i, s_{i+1}) > 0$ for all $i \in \mathbb{N}$. A finite path $\rho = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_n$ is a prefix of an infinite path ending in a state. We denote by $FPath_\mathcal{M}$ and $IPath_\mathcal{M}$, respectively, the set of all finite and infinite paths of $\mathcal{M}$ starting from state $\overline{s}$.

The choice of action to take at each step of the execution of an MDP $\mathcal{M}$ is made by a *policy* (sometimes also referred to as a strategy or scheduler), which can base its decision on the history of $\mathcal{M}$ up to the current state. Formally, a policy is a function $\pi : FPath_\mathcal{M} \to A$ such that, for any finite path σ ending in state s_n, $\pi(\sigma) \in A(s_n)$. Important classes of policy include those that are *memoryless* (which only base their choice on the current state) and *finite-memory* (which need to track only a finite set of "modes").

Given an MDP $\mathcal{M}$ and a policy π for it, we can define a probability measure $Pr^\pi_\mathcal{M}$ over the set of infinite paths $IPath_\mathcal{M}$, which allows us to determine the probability with which certain events occur under policy π. We can also use *expected values* $E^\pi_\mathcal{M}(\cdot)$, for example of a cost function over paths. A useful example is the following problem.

Problem 1: Let $\mathcal{M}$ be an MDP and $p \in AP$ an atomic proposition. Generate a policy $\pi^{min}_\mathcal{M}(c, \mathtt{F}\, p)$ for $\mathcal{M}$ that minimises the expected value of the accumulated cost to reach a state $s \in S$ such that $p \in Lab(s)$. More formally, let us consider the accumulated cost function $cumul(c, \mathtt{F}\, p) : IPath_\mathcal{M} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ defined as:

$$cumul(c, \mathtt{F}\, p)(s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots) = \begin{cases} \infty & \text{if } p \notin Lab(s_j)\, \forall j \in \mathbb{N} \\ \sum_{j=0}^{k_p - 1} c(s_j, a_j) & \text{otherwise,} \end{cases}$$

where $k_p = \min\{k \in \mathbb{N} \mid p \in Lab(s_k)\}$. The expected accumulated cost to reach a p-labelled state under a policy π is the expected value $E^\pi_\mathcal{M}(cumul(c, \mathtt{F}\, p))$. Thus, our aim is to generate a policy $\pi^{min}_\mathcal{M}(c, \mathtt{F}\, p)$ such that:

$$\pi^{min}_\mathcal{M}(c, \mathtt{F}\, p) = \arg\min_\pi E^\pi_\mathcal{M}(cumul(c, \mathtt{F}\, p)).$$

Problem 1 can be solved using standard MDP algorithms such as value or policy iteration [9], after first performing a graph analysis of the model to identify states for which the cost is infinite [3]. These techniques are supported by the PRISM tool, on which we base our implementation. In Section III, we will generalise this approach to consider the expected accumulated cost of satisfying co-safe LTL formulas, rather than simple reachability properties.

C. Linear Temporal Logic

Linear temporal logic (LTL) is an extension of propositional logic which allows us to reason about infinite sequences of states. It was developed as a means for formal reasoning about concurrent systems [8], and provides a convenient and powerful way to formally specify a variety of qualitative properties of a system. We define LTL formulas φ over propositions AP using the following grammar:

$$\varphi ::= true \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathtt{X}\, \varphi \mid \varphi\, \mathtt{U}\, \varphi, \text{ where } p \in AP$$

The $\mathtt{X}$ operator is read "next", meaning that the formula it precedes will be true in the next state. The $\mathtt{U}$ operator is read "until", meaning that its second argument will eventually become true in some state, and the first argument will be continuously true until this point. See, for example, [8] for the formal semantics of the logic. Other useful LTL operators can be derived from the ones above. A common example is the "eventually" operator $\mathtt{F}\, \varphi$, which requires that φ is satisfied in some future state: $\mathtt{F}\, \varphi \equiv true\, \mathtt{U}\, \varphi$.

D. Co-safe LTL

We now introduce the class of LTL formulas that we will use to specify tasks. We are interested in minimising the expected accumulated cost to achieve some task. However, this is not well defined if the task is an arbitrary LTL formula since, in general, the accumulated cost may diverge. Thus, we will restrict ourselves to LTL formulas that can be "satisfied" in a finite horizon. This is a well-defined class, called *co-safe* formulas. These are formulas for which the satisfying infinite sequences always have a finite *good prefix*. Given an LTL formula φ and $w = w_0 w_1 \dots \in (2^{AP})^\omega$ such that $w \vDash \varphi$, we say w has a good prefix if there exists $n \in \mathbb{N}$ for which the truncated finite sequence $w|_n = w_0 w_1 \dots w_n$ is such that $w|_n \cdot w' \vDash \varphi$ for any infinite sequence $w' \in (2^{AP})^\omega$. We also define the length of the shortest good prefix of w for φ:

$$k^w_\varphi = \min\{k \in \mathbb{N} \mid w_0 w_1 \dots w_k \text{ is a good prefix for } \varphi\}$$

We say that the LTL formula φ is co-safe if all the infinite sequences π such that $\pi \models \varphi$ have a good prefix. It is known [5] that an LTL formula which is in positive normal form (i.e., where negation is only applied directly to atomic propositions) and which only uses the temporal operators X, U and F is co-safe. We will keep our formulas within this syntactic restriction.

We will be translating co-safe LTL to deterministic finite automata (DFAs). A DFA is a tuple $\mathcal{A} = \langle Q, \overline{q}, Q_F, \Sigma, \delta_{\mathcal{A}} \rangle$, where: (i) Q is a finite set of states; (ii) $\overline{q} \in Q$ is the initial state; (iii) $Q_F \subseteq Q$ is the set of accepting states; (iv) Σ is a finite alphabet; and (v) $\delta_{\mathcal{A}} : Q \times \Sigma \to Q$ is a (partial) deterministic transition function.

We extend the transition function, in standard fashion, by inductively defining $\delta^+_{\mathcal{A}} : Q \times \Sigma^* \to Q$ such that $\delta^+_{\mathcal{A}}(q, \epsilon) = q$ and $\delta^+_{\mathcal{A}}(q, w.\alpha) = \delta_{\mathcal{A}}(\delta^+_{\mathcal{A}}(q, w), \alpha)$ for $w \in \Sigma^*$ and $\alpha \in \Sigma$. The language accepted by $\mathcal{A}$ is then defined as:

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^+_{\mathcal{A}}(\overline{q}, w) \in Q_F\}$$

It is known [5] that, for any co-safe LTL formula φ written over AP, we can build a DFA $\mathcal{A}_\varphi = \langle Q, \overline{q}, Q_F, 2^{AP}, \delta_{\mathcal{A}_\varphi} \rangle$ that accepts exactly the good prefixes for φ. Furthermore, given that a good prefix can be "completed" in any way and will still satisfy φ, an accepting state $q_F \in Q_F$ is such that $\delta_{\mathcal{A}_\varphi}(q_F, \alpha) = q_F$ for all $\alpha \in 2^{AP}$.

III. Planning for MDPs with Co-safe LTL

We now show how to generate policies that minimise the accumulated cost to satisfy a co-safe LTL formula, by extending the simpler case of the expected cost to reach a target (see, e.g., [3]). The problem can be posed as follows.

Problem 2: Let $\mathcal{M}$ be an MDP and φ a co-safe LTL formula over AP. Generate a policy $\pi^{min}_{\mathcal{M}}(c, \varphi)$ for $\mathcal{M}$ minimising the expected value of the accumulated cost to generate a path $s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{k-1}} s_k \in FPath_{\mathcal{M}}$ that is a good prefix for φ. More formally, consider the accumulated cost function $cumul(c, \varphi) : IPath_{\mathcal{M}} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ defined as:

$$cumul(c, \varphi)(s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots) = \begin{cases} \infty & \text{if } Lab(s_0)Lab(s_1)\ldots \not\models \varphi \\ \sum_{j=0}^{k_\varphi^{Lab(s_0)Lab(s_1)\ldots}} c(s_j, a_j) & \text{otherwise.} \end{cases}$$

The expected value of $cumul(c, \varphi)$ for MDP $\mathcal{M}$ under policy π is $E^\pi_{\mathcal{M}}(cumul(c, \varphi))$, and our aim is to generate a policy $\pi^{min}_{\mathcal{M}}(c, \varphi)$ such that:

$$\pi^{min}_{\mathcal{M}}(c, \varphi) = \arg\min_\pi E^\pi_{\mathcal{M}}(cumul(c, \varphi))$$

To find such a policy, we reduce the problem to one on an MDP-DFA *product*. More precisely, given an MDP $\mathcal{M}{=}\langle S, \overline{s}, A, \delta_{\mathcal{M}}, AP, Lab, c\rangle$, co-safe LTL formula φ and a corresponding DFA $\mathcal{A}_\varphi{=}\langle Q, \overline{q}, Q_F, 2^{AP}, \delta_{\mathcal{A}_\varphi}\rangle$ the product of $\mathcal{M}$ and $\mathcal{A}_\varphi$, denoted $\mathcal{M} \otimes \mathcal{A}_\varphi$, is given by the MDP $\mathcal{M} \otimes \mathcal{A}_\varphi = \langle S \times Q, (\overline{s}, q_\varphi), A, \delta_{\mathcal{M}_\varphi}, AP, Lab_\varphi, c_\varphi\rangle$, where:

- $q_\varphi = \delta_{\mathcal{A}_\varphi}(\overline{q}, Lab(\overline{s}))$
- $\delta_{\mathcal{M}_\varphi}((s, q), a, (s', q')) = \delta_{\mathcal{M}}(s, a, s')$ if $q' = \delta_{\mathcal{A}_\varphi}(q, Lab(s'))$, and 0 otherwise;
- $Lab_\varphi((s, q)) = Lab(s) \cup \{acc_\varphi\}$ if $q \in Q_F$, and $Lab(s)$ otherwise;
- $c_\varphi((s, q), a) = c(s, a)$.

For clarity, we will also denote this product MDP by $\mathcal{M}_\varphi$, i.e., $\mathcal{M}_\varphi = \mathcal{M} \otimes \mathcal{A}_\varphi$. Intuitively, the MDP $\mathcal{M}_\varphi$ behaves exactly like the original MDP $\mathcal{M}$, but is augmented with information about the satisfaction of φ. Once a path of $\mathcal{M}_\varphi$ reaches an *accepting state* (i.e. one labelled with the new atomic proposition acc_φ), it satisfies the formula φ. Furthermore, as stated before, since we assume that φ is a co-safe LTL formula, all subsequent states along such a path will also be accepting.

It is known [1] that the product MDP $\mathcal{M}_\varphi$ preserves the probabilities of paths from $\mathcal{M}$. Since we also copy the cost function c of $\mathcal{M}$ directly to $\mathcal{M}_\varphi$, the following holds:

$$E^{min}_{\mathcal{M}}(c, \varphi) = E^{min}_{\mathcal{M}_\varphi}(c_\varphi, \mathtt{F}\, acc_\varphi)$$

Thus, to solve Problem 2, it suffices to find a policy that minimises the expected cost to reach such an accepting state in the product MDP, which reduces to solving Problem 1 for $\mathcal{M}_\varphi$. As a final step, we need to convert an optimal policy obtained for $\mathcal{M}_\varphi$ to one for the original MDP $\mathcal{M}$. As described before, finding a policy that minimises the cost of reaching a state labelled with acc_φ in $\mathcal{M}_\varphi$ can be done with standard MDP techniques such as value or policy iteration. In fact, the result of this is a *memoryless* policy, which takes the form $\pi : S \times Q \to A$ (since the state space of $\mathcal{M}_\varphi$ is $S \times Q$). Using this policy and $\mathcal{M}_\varphi$, we can construct a *finite-memory* policy of $\mathcal{M}$ for Problem 2. Intuitively, the elements q of states (s, q) in the product MDP represent "modes" of the finite-memory policy, which keep track of the path executed by the MDP so far, and $\pi(s, q)$ gives the action to take in state s for mode q. Precise details of a similar policy construction (for probabilistic safety properties) can be found in [3].

We have extended the PRISM software to generate these policies, and will be using it in our implementation.

IV. Dynamic Replanning During Execution

In this section, we tackle the problem of new tasks arriving during the execution of a policy. When a new task arrives, we want to create a new policy that minimises the expected value of the accumulated cost of satisfying both task(s) already being executed and the new task. This policy should also take into account the extent to which the current tasks have been executed so far. The generation and execution of such policies is implemented by Algorithms 1 and 2.

Given a new LTL task φ, Algorithm 1 creates a new product MDP, which is then used to synthesise a policy that takes φ into account. This new MDP is based on the product composition presented in the previous section. In order to keep track of when each task φ_i has been completed, we have an atomic proposition acc_{φ_i} in the product MDP. Furthermore, in order to be able to minimise the expected accumulated cost to satisfy the conjunction of all tasks, we add a new atomic proposition $goal$ to states in the MDP

Algorithm 1 DYN_REPLAN($\{\varphi_1, ..., \varphi_n\}, \mathcal{M}_{\varphi_1 \wedge ... \wedge \varphi_n}, \varphi$)

Input: Current task list $\{\varphi_1, ..., \varphi_n\}$, current product MDP $\mathcal{M}_{\varphi_1 \wedge ... \wedge \varphi_n}$, new task φ.

Output: New product MDP $\mathcal{M}_{\varphi_1 \wedge ... \wedge \varphi_n \wedge \varphi}$, optimal policy $\pi^{min}_{\mathcal{M}}(c, \varphi_1 \wedge ... \wedge \varphi_n \wedge \varphi)$

1: $\mathcal{M}_{\varphi_1 \wedge ... \wedge \varphi_n \wedge \varphi} \leftarrow \mathcal{M}_{\varphi_1 \wedge ... \wedge \varphi_n} \otimes \mathcal{A}_\varphi$ (let $\mathcal{M}_{\varphi_1 \wedge ... \wedge \varphi_n \wedge \varphi} = \langle S', \overline{s}', A, \delta_{\mathcal{M}'}, AP', Lab', c' \rangle$)
2: **for all** $s \in S'$ **do**
3: **if** $\left(\bigcup_{i=1,...,n} \{acc_{\varphi_i}\}\right) \cup \{acc_\varphi\} \subseteq Lab'(s)$ **then**
4: $Lab'(s) \leftarrow Lab'(s) \cup \{goal\}$
5: **end if**
6: **end for**
7: **return** $(\mathcal{M}_{\varphi_1 \wedge ... \wedge \varphi_n \wedge \varphi}, \pi^{min}_{\mathcal{M}_{\varphi_1 \wedge ... \wedge \varphi_n \wedge \varphi}}(c', \mathtt{F}\ goal))$

that are labelled as accepting states for *all* the tasks that are currently being executed.

The states of a product MDP constructed for multiple tasks $\varphi_1, \ldots, \varphi_n$ are of the form $(s, q_1, \ldots, q_n)$, where s is the state of the system being controlled (e.g. a robot) and each q_i is a state in the DFA $\mathcal{A}_{\varphi_i}$, which can be seen as representing the extent to which task φ_i has been completed. The initial state of the current product MDP thus tells us the current status of each task. When we construct a new product MDP after the arrival of a new task, we generate the reachable fragment of this MDP, using the current initial state as a starting point. This means that the we can replan dynamically, whilst preserving the current status of all existing tasks.

We also note that, since tasks are specified as co-safe LTL formulas, we are guaranteed that, once an accepting state for a given task is reached, all states reachable from that one are also accepting for the task. Thus, when we call Algorithm 1, tasks that have been completed since the last new task arrived are automatically deleted from the product MDP. This occurs because we only construct the reachable fragment of the MDP from the current state.

Algorithm 2 is in charge of executing the policies and replanning when new tasks are specified. It starts with the MDP model of the system and an initial task φ_{init}, and creates a product MDP and a policy for this input. It then enters a while loop (line 3), until all tasks are satisfied. It starts by executing the action given by the current optimal policy (line 4). While an action is being executed, new tasks can be given to the system. If this happens, replanning is needed. Thus, we delete the *goal* labels from all states, and call Algorithm 1 for the current product MDP and the new LTL task (lines 6 – 11).

After an action is executed, we check the new state of the system and evolve the state of the product MDP accordingly. In order to keep track of the status of execution of the current task, we change the initial state of the product MDP to the state that was reached (line 14). Then, we check if any task was completed in the new state. If so, we delete it from the list of current tasks and delete the atomic proposition that represents its accepting states from the product MDP

Algorithm 2 DYN_EXECUTE($\mathcal{M}$, φ_{init})

Input: Initial MDP $\mathcal{M}$, initial task φ_{init}

1: $(\mathcal{M}_{cur}, \pi_{cur}) \leftarrow$ DYN_REPLAN($\emptyset, \mathcal{M}, \varphi_{init}$) (let $\mathcal{M}_{cur} = \langle S_{cur}, \overline{s}_{cur}, A, \delta_{\mathcal{M}_{cur}}, AP_{cur}, Lab_{cur}, c_{cur} \rangle$)
2: $tasks \leftarrow \{\varphi_{init}\}$
3: **while** $goal \notin Lab_{cur}(\overline{s}_{cur})$ **do**
4: execute policy action $\pi_{cur}(\overline{s}_{cur})$
5: **while** $\pi_{cur}(\overline{s}_{cur})$ is being executed **do**
6: **if** new task φ arrives **then**
7: **for all** $s \in S_{cur}$ **do**
8: $Lab_{cur}(s) = Lab_{cur}(s) \setminus \{goal\}$
9: **end for**
10: $(\mathcal{M}_{cur}, \pi_{cur}) \leftarrow$ DYN_REPLAN($tasks, \mathcal{M}_{cur}, \varphi$)
11: $tasks \leftarrow tasks \cup \{\varphi\}$
12: **end if**
13: **end while**
14: update $\overline{s}_{cur}$ to next state reached in $\mathcal{M}_{cur}$ after executing $\pi_{cur}(\overline{s}_{cur})$
15: **for all** $\varphi \in tasks$ **do**
16: **if** $acc_\varphi \in Lab_{cur}(\overline{s}_{cur})$ **then**
17: $tasks \leftarrow tasks \setminus \{\varphi\}$
18: $AP \leftarrow AP \setminus \{acc_\varphi\}$
19: **for all** $s \in S_{cur}$ **do**
20: $Lab_{cur}(s) = Lab_{cur}(s) \setminus \{acc_\varphi\}$
21: **end for**
22: **end if**
23: **end for**
24: **end while**

(lines 15 – 24). Note that, as we stated before, we do not need to trim the states of the MDP because the next time Algorithm 1 is called, we will create a product MDP in which information related to the completed co-safe task will automatically disappear from the resulting structure.

Finally, we remark that we are assuming the LTL specifications are not conflicting, i.e., that their conjunction is satisfiable in the MDP model. Checking this can be easily done – if no policy can be generated, then the new task conflicts with one of the current tasks – but we chose not to explicitly add this to our description due to lack of space.

V. APPLICATION TO MOTION PLANNING

In this section, we describe an implementation of our approach to a motion planning scenario for a mobile robot. This application example was implemented on a *MetraLabs Scitos A5* robot running on a public area of the School of Computer Science building of the University of Birmingham. In Fig. 1, we show a photograph of the robot in its environment, along with the map and navigation graph used.

In the navigation graph, each node corresponds to a different position of the robot in the environment, and edges represent navigation actions between nodes. The continuous navigation between nodes was provided by the ROS navigation stack[1]. We created an MDP model for this graph,

[1] `http://wiki.ros.org/navigation`

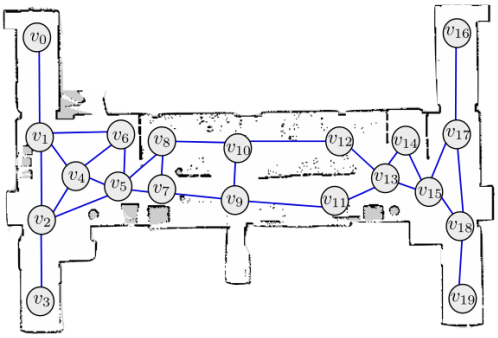

Fig. 1. The robot in its environment, and the map and navigation graph used in the application example. Blue (bi-directional) edges represent possible navigation actions between states.

where each state is labelled by an atomic proposition v_i, which corresponds to the navigation node that the state is representing. We also take into account possible failures in navigation. In this example, we consider that a failure occurs when the robot fails to reach the target node of the navigation action, for example due to an obstacle, and ends in a different node. We model these failures by adding uncertainty to the outcome of executing actions from some states. For example, action $goto_{11}$ from state v_{13} has probability 0.85 of ending in state v_{11}, 0.1 of ending in state v_{12} and 0.05 of finishing in state v_{14}. In order to define a cost function for the MDP, we used the Euclidean distance between nodes.

For the execution of the policies obtained from our approach, we used the Markov Decision Making library[2] for ROS. In Fig. 2, we depict different moments in the execution of our algorithms for 3 co-safe LTL tasks specified dynamically during execution. The robot starts in node v_0 with the task "visit v_3 and v_{18}, in any order", i.e., $\mathtt{F}\, v_3 \wedge \mathtt{F}\, v_{18}$. Algorithm 1 creates a finite-memory policy for this task and the robot executes it, navigating towards v_3 first, as depicted in Fig. 2(a). Note that we have an optimal action defined for each state, thus the choice of first node to be visited depends on the current state of the robot. This means that even if there are action failures, there is no need for replanning. When the robot reaches v_3, the "mode" of the policy changes, and the optimal actions for each state are now directed towards node v_{18}, as seen in Fig. 2(b). Recall that the "mode" changes are due to a change of one of the DFA state components in the evolution of the MDP-DFA product.

While the robot is executing action $goto_{11}$ from state v_9, we add a new task: "visit v_9 and afterwards visit v_{14}", i.e., $\mathtt{F}(v_9 \wedge \mathtt{F}\, v_{14})$[3]. The dynamic replanning is executed, and a new policy is generated. This policy takes into account that we still need to visit v_{18}, but also incorporates the fact that v_9 needs to be visited. Since the robot is closer to v_9, it turns back to visit it. This is seen in Fig. 2(c). After v_9 is visited, the policy changes "mode" again, now taking into account the fact that v_{14} needs to be visited after v_9, and that v_{18} is still to be visited. The shortest path at this moment is moving towards node v_{14}, so the robot moves towards it (Fig. 2(d)).

When the robot reaches v_{14}, we add a new task: "visit v_0, avoiding v_8", i.e., $\neg v_8\, \mathtt{U}\, v_0$. In practical terms, such specifications, where given nodes are to be avoided, can be used when it is known that a given area of the environment is not safe, for example due to the presence of a crowd. If this information is known beforehand it can be added to the specification in order to prevent navigation problems that might occur. With this new specification, a new policy is computed. Node v_0 becomes a node to be visited, and node v_8 a node to be avoided. However, given that the current position of the robot is closer to v_{18}, the policy drives the robot towards it, as seen in Fig. 2(e).

Finally, when the robot reaches v_{18} the policy changes "mode", and starts driving the robot towards v_0. However, when trying to execute action $goto_{11}$ from v_{13}, an obstacle makes the robot's continuous navigation end on v_{12} instead. Given that the optimal action from v_{12} is $goto_{10}$, the robot switches from its initial most expected trajectory (through v_{11}) to a new one, which is the optimal given the navigation failure. After that, given that v_8 is a forbidden node, the policy makes the robot turn and avoid it, finally getting to v_0 and finishing execution, as all the LTL tasks have been completed (Fig. 2(f)).

In Table I, we show, for the addition of each task described above, the number of states and transitions of the current product MDP, along with the computation time of the new optimal policy[4]. We see that, for this small example, the computation times are negligible. Furthermore, keeping track of the current state of execution and only taking into account the reachable fragment from the current state of the product MDP when replanning keeps the size of the structures from increasing greatly. To illustrate this fact, we also show the size and computation time for the case where the initial task is the conjunction of all 3 tasks used in the example.

[2] `https://github.com/larsys/markov_decision_making`

[3] One could also make sure that v_{14} cannot be visited before state v_9 by changing the specification to $(\neg v_{14}\, \mathtt{U}\, v_9) \wedge \mathtt{F}\, v_{14}$.

[4] This includes building the DFA, building the product MDP, and finding the optimal policy. All computations were performed on an Intel® Core™ i7 quad-core CPU at 2.20GHz and 8GB of RAM.

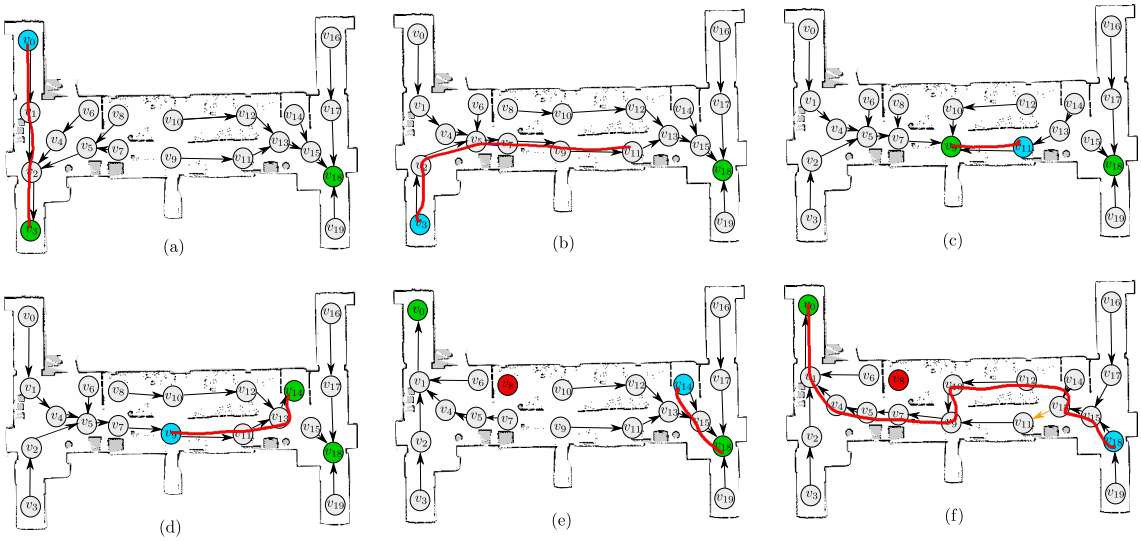

Fig. 2. Depiction of the different policies at different moments of execution. States in blue are the current state of the robot at each time, states in green need to be visited to fulfil the global LTL task, and states in red need to be avoided. The continuous trajectory taken by the robot is depicted in red, while the optimal action for each state in each given moment (given by the policy obtained by running Algorithms 1 and 2) is represented by arrows between states. Arrows in orange represent failures while performing that action.

TABLE I

MDP SIZES AND COMPUTATION TIMES FOR ADDITION OF TASKS.

Task added	$\lvert S \rvert$	$\lvert \delta_{\mathcal{M}} \rvert$	**Time (s)**
–	20	56	–
$\varphi_1 = \mathrm{F}\, v_3 \wedge \mathrm{F}\, v_{18}$	75	215	0.05
$\varphi_2 = \mathrm{F}(v_9 \wedge \mathrm{F}\, v_{14})$	110	314	0.06
$\varphi_3 = \neg v_8 \,\mathrm{U}\, v_0$	110	316	0.06
$\varphi_1 \wedge \varphi_2 \wedge \varphi_3$	394	1160	0.29

VI. CONCLUSIONS

We have presented an approach to generate cost-optimal policies for MDPs, with goals given as co-safe LTL formulas. We also presented a dynamic execution and replanning procedure, where new co-safe LTL tasks can be given to the system during execution. This methodology was illustrated by an example where optimal motion plans for a mobile robot were generated. As illustrated, the presented approach inherently tackles the uncertainty associated with action execution by robot systems, and provides the user with a flexible way of specifying tasks. Future work includes defining a more refined MDP model for navigation, and developing an algorithm to learn both the transition failure probabilities and the expected times between nodes from real data gathered by the robot. We will also integrate this approach with a scheduler with notion of real time, so that a user can allocate navigation tasks to different times of day. We then plan to create an interface where users can schedule tasks for a mobile robot which is in continuous execution in a human populated environment. Regarding policy generation, we also plan to extend its capabilities by, for example, adding reactivity to sensor readings, investigating the use of other classes of LTL, and exploring the possibility of using multi-objective specifications for MDPs.

VII. ACKNOWLEDGEMENTS

The authors would like to thank João Messias for his help with the Markov Decision Making library. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement No 600623, STRANDS, and the EPSRC grant EP/K014293/1.

REFERENCES

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

[2] X. C. Ding, S. L. Smith, C. Belta, and D. Rus. Optimal control of Markov decision processes with linear temporal logic constraints. *IEEE Transactions on Automatic Control*, 59(5):1244–1257, 2014.

[3] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In *Formal Methods for Eternal Networked Software Systems (SFM'11)*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011.

[4] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Temporal logic-based reactive mission and motion planning. *IEEE Transactions on Robotics*, 25(6):1370–1381, 2009.

[5] O. Kupferman and M. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[6] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[7] B. Lacerda and P. U. Lima. Designing Petri net supervisors from LTL specifications. In *Proc. of Robotics: Science and Systems VII*, 2011.

[8] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[9] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

[10] M. Svoreňová, I. Černá, and C. Belta. Optimal control of MDPs with temporal logic constraints. In *Proc. of CDC'13: 52nd IEEE Conf. on Decision and Control*, 2013.

[11] A. Ulusoy, T. Wongpiromsarn, and C. Belta. Incremental control synthesis in probabilistic environments with temporal logic constraints. In *Proc. of CDC'12: IEEE Conf. on Decision and Control*, 2012.

[12] E. M. Wolff, U. Topcu, and R. M. Murray. Optimal control with weighted average costs and temporal logic specifications. In *Proc. of Robotics: Science and Systems VIII*, 2012.

[13] E. M. Wolff, U. Topcu, and R. M. Murray. Efficient reactive controller synthesis for a fragment of linear temporal logic. In *Proc. of ICRA '13: IEEE Int. Conf. on Robotics and Automation*, 2013.