An Analysis of Controller Synthesis for Systems with Uncertainty over a Subset of Linear Temporal Logic

ECE1653
Dec. 13, 2018
Yuan Liu

1 Abstract

In this report, controller synthesis for system with uncertainty, based on Linear Temporal Logic, is explored by a literature study mainly based on the paper, Optimal Control of Non-deterministic Systems for a Computationally Efficient Fragment of Temporal Logic, by Wolff et al.[2]. In addition, feasible policy synthesis procedure is analyzed and implemented in code for demonstration. The proposed framework by the authors in [2] offers an alternative approach in feasible and optimal synthesis based on a subset of LTL which bypasses use of a full fledged automaton during synthesis and reduces high computational complexity. Feasibility of a system with uncertainty is reduced to feasibility under worst-case conditions of non-determinism with regard to LTL propositions. Ideas behind the proposed framework is first analyzed. Then, feasibility algorithm is implemented and used to demonstrate feasible controller synthesis for several scenarios. Simulations for these scenarios with their respective synthesized controllers are performed and sample runs are recorded for analysis. Optimal task policy synthesis is briefly discussed but not implemented for demonstration. Finally, limitations and analysis of simulations are discussed for improvement.

2 Investigation

A set of notations used throughout the paper is briefly described in the following section. This is followed by a high level overview of the strategy taken by the authors in controller synthesis. Finally, algorithmic implementation of synthesis procedure as proposed by the authors is discussed.

2.1 Notation

The symbol conventions follow those in authors' work [1]. A list of frequently used notations are briefly summarized below.

2.1.1 LTL Fragment

One of the main ideas behind the papers, [3] and [2], is constrained usage of LTL in order to reduce computational complexity of producing controllers while retaining meaningful and expressive rules for the type of controller the process can create. In particular, [2] presents a synthesis solution based on a LTL subset of the form,

$$\varphi = \varphi_{safe} \wedge \varphi_{resp} \wedge \varphi_{resp}^{ss} \wedge \varphi_{per} \wedge \varphi_{task} \tag{1}$$
$$\varphi_{safe} := \Box \varphi_s$$
$$\varphi_{resp} := \bigwedge_{j \in I_r} \Box(\psi_{r,j} \implies \bigcirc \phi_{r,j})$$
$$\varphi_{resp}^{ss} := \bigwedge_{j \in I_{sr}} \Diamond\Box(\psi_{sr,j} \implies \bigcirc \phi_{sr,j})$$
$$\varphi_{per} := \Diamond\Box\psi_p$$
$$\varphi_{task} := \bigwedge_{j \in I_t} \Box\Diamond\psi_{t,j}$$

. In particular, φ denote a propositional formula. φ_{safe}, φ_{resp}, φ_{resp}^{ss}, φ_{per}, φ_{task} denote safety constraint, next-step response, steady-state next-step response, persistence, recurring tasks respectively. $I_s, I_r, I_{sr}, I_p, I_t$ are index sets of different types of propositions. The above form (1) is a subset of LTL because certain forms such as disjunctions, $\Diamond\psi$, and $\Box(\psi \implies \Diamond\psi)$ are not included.

Examples of systems using the above LTL fragment are shown in section 3.

2.1.2 Transition System

Modelling of the system with uncertainty is done using a non-deterministic transition system (NTS) is introduced with the notation,

$$\mathcal{T} = (S, A, R, s_0, AP, L, c) \tag{2}$$

, where S is the set of state, R is transition function: $(S, A) \mapsto 2^S$, s_0 is the initial state, AP is the set of atomic propositions, L is the labeling function: $S \mapsto 2^{AP}$, c is the cost function: $(S, A, S) \mapsto \mathbb{R}$. This is very similar to deterministic transition system, with the difference being extension of transition and labelling functions having multiple elements in their codomain. The exact topology of the input transition system that we start with has to be tailored to the specific problem which may include continuous time domain reachability constraints. The proposed framework operates on a discretized time domain and assumes there is suitable conversion between continuous and discrete domains if applicable.

The subset of LTL, (1), is accepted by the authors' synthesis procedure in order to avoid high computation cost where the complexity for determining feasibility for non-deterministic system is reduced from doubly exponential ($\mathcal{O}(2^{2^{|\varphi|}})$) when computing over full LTL using available software tool designed for Generalized Rabin(1) automaton to $\mathcal{O}(|\varphi||F_{min}|(|S|+|R|))$ when computing over the fragment (1) where F_{min} is a set of favourable states in order to satisfy φ_{task} and this is further discussed in the following subsections. In addition, several different variations of cost function for determining optimal task ordering as proposed by the authors are briefly described in later sections.

2.1.3 Value Function

Value function is used throughout the authors' definition and implementation of controller synthesis for optimality. It is denoted in recursive form as

$$V^c_{\mathcal{B},\mathcal{T}}(s) = \underset{a\in A(s)}{min} \underset{t\in R(s,a)}{max} V^c_{\mathcal{B},\mathcal{T}}(t) + c(s,a,t) \qquad (3)$$

, where it is parameterized by transition system $\mathcal{T}$, destination set B, and transition cost function c. This is generally used in computing the controlled predecessor set which contains states, of computed value to be $< \infty$, that are able to reach B under a policy that is under agent's control.

A variant, forced value function, is denoted by

$$V^f_{\mathcal{B},\mathcal{T}}(s) = \underset{a\in A(s)}{max} \underset{t\in R(s,a)}{max} V^f_{\mathcal{B},\mathcal{T}}(t) + c(s,a,t) \qquad (4)$$

. This is generally used in forced predecessor set to compute a set of states whose values are $< \infty$ (B initialized to 0 and $S\backslash B$ initialized to ∞) which indicate that they are guaranteed to reach B under worst-case conditions of non-determinism.

2.1.4 Winning Set

A winning set, W, of a transition system, $\mathcal{T}$, is defined as the set of states where there exists some policy, μ, where all possible runs σ of NTS starting at those states satisfy the given proposition φ.

$$W := \{s \in S \mid (\exists\mu)(\forall\sigma = \mathcal{T}^{\mu}(s))(\sigma \models \varphi)\} \qquad (5)$$

2.1.5 Predecessor

Controlled predecessor set is defined as

$$CPred^{\infty}_{\mathcal{T}}(B) := \{s \in S \mid V^c_{B,\mathcal{T}}(s) < \infty\} \qquad (6)$$

and forced predecessor set is defined as

$$FPred^{\infty}_{\mathcal{T}}(B) := \{s \in S \mid V^f_{B,\mathcal{T}}(s) < \infty\} \qquad (7)$$

, where ∞ denotes unreachability for the associated state with respect to B.

2.2 Strategy

System modelling is abstracted in terms of transition systems and feasible actions are determined through use of value functions and reachability computations. Specifically, system interaction is decomposed of actions by the agent (of which we control) and the environment, which can be regarded as an adversary because feasibility synthesis is done by considering the worst-case scenario. Thus, it can be thought as a parity game and feasibility synthesis relates to a minimax solution. The proposed strategy in [2] for solving the controller synthesis for the LTL fragment (1) is divided into feasibility and optimality and they are outlined below.

2.2.1 Feasibility

1. Determining the existence of the winning set, W, for tasks related to φ_{task} with imposed constraints, φ_{safe}, φ_{resp}, φ_{per} and φ^{ss}_{resp}. Return it if it exists.

2. If W exists, compute the predecessor set for W while satisying φ_{safe} and φ_{resp}, and if it coincides with s_0, then a feasible policy is computed to go from s_0 to W. Otherwise, a feasible controller does not exist.

3. If W exists and the predecessor set for W coincides with s_0 as computed earlier, compute feasible policies for reaching each task set in φ_{task} with the relevant constraints. These policies are further controlled (switched) by a finite memory controller to be visited repeatedly.

Item 1 results from the observation that φ is satisfied by visiting a set of states coupled with some control policy that enables the agent to produce all runs from NTS which satisfy φ. The returned winning set, W are the favourable states that the agent would want to reach because all runs starting in each of those favourable states satisfy φ.

Item 2 determines a policy for going from s_0 to a state in W. This can be regarded as a transient phase where the agent traverses through obstacles and settle into the steady state region where it would remain indefinitely.

Item 3 computes feasible policies to reach recurring task states in the steady state phase of a run. The exact ordering of the recurring tasks can be further optimized by a given task cost optimizer. Since the run is infinite and tasks are recurrent, the task cost function is usually a variation of minimization of average traversal cost or minimization of longest traversal between task states.

2.2.2 Optimality

A predefined task cost function is selected and an optimal ordering for the recurring tasks is computed, usually with the help of a directed graph. Only recurring tasks are considered in computation of cost since optimization is intended for infinite amount of time. The directed graph for ordering is computed on a subset of the original graph and the optimization is done for the cost of traversing between nodes of the directed graph.

When constructing a directed graph, additional nodes and edges have to be created to account for the difference of non-deterministic path costs. However, it is suggested that approximating these subset nodes related to non-deterministic paths with a single node, as the case for deterministic transition system, can reduce computational complexity. This can be done purely for performance tradeoff.

Different cost functions such as average cost per task cycle, minimax, and average cost can be used for shaping optimal policies and these are computed over the constructed graph. They are further explored in the Algorithm section.

2.3 Algorithm

The authors' proposed framework in which algorithms for synthesizing controllers for NTS and LTL fragment (1) is briefly summarized below.

2.3.1 Top Level Procedure

The overall synthesis procedure, *alg.* (1), from the authors' paper, [2], is summarized below.

From line 2 to 5 of *alg.* (1), several transition systems are constructed. These are operated on by submodules which compute reachability between different sets of state space. $\mathcal{T}_{resp}$ is created by retaining transitions as satisfied by φ_{resp}. $\mathcal{T}_{safe}$ is created by pruning transitions to a set of states which are guaranteed to violate φ_{resp}, assuming worst-case non-determinism. This means under worst-case adversarial actions by non-determinism of the system, there is no policy that can avoid reaching states that violate φ_{resp}. $\mathcal{T}_{per}$ is created by pruning transitions to a set of states which are guaranteed to violate φ_{per}, assuming worst-case non-determinism. $\mathcal{T}^{ss}_{resp}$ is created by retaining transitions as satisied by φ^{ss}_{resp}. Each of these transition system filters and prunes on the associated input transition system and hence each one is a more constrained version of the previous, due to LTL fragment specifications.

Algorithm 1: Top Level Controller Synthesis

```
input  : $\mathcal{T}$, $\varphi_{safe}$, $\varphi_{resp}$, $\varphi^{ss}_{resp}$, $\varphi_{per}$, $\varphi_{task}$, c, $c_{tasks}$, $I_t$
output: optimal controller with memoryless policies for $s_0$ to steady state region, a set of memoryless policies for each task region, an optimized ordered set of task indicies to switch in a modolu fashion
1  begin
2      $\mathcal{T}_{resp} \leftarrow \mathcal{T}|_{\varphi_{resp}}$ //filter, pruning stage
3      $\mathcal{T}_{safe} \leftarrow \mathcal{T}_{resp}|_{S-FPred^{\infty}_{\mathcal{T}_{resp}}(S-[[\varphi_{safe}]])}$
4      $\mathcal{T}_{per} \leftarrow \mathcal{T}_{safe}|_{S-FPred^{\infty}_{\mathcal{T}_{safe}}(S-[[\varphi_{per}]])}$
5      $\mathcal{T}^{ss}_{resp} \leftarrow \mathcal{T}_{per}|_{\varphi^{ss}_{resp}}$
6      $W, F \leftarrow$taskFeasibility($\mathcal{T}^{ss}_{resp}$, $\varphi_{task}$)
7      $W_{predec} \leftarrow CPred^{\infty}_{\mathcal{T}_{safe}}(W)$
8      if $W_{predec} \cap s_0 = \{\emptyset\}$
9          return ($\emptyset$, $\emptyset$, $\emptyset$) // infeasible
10     $\mathcal{P}_W \leftarrow$ actions induced by $V^c_{W,\mathcal{T}_{safe}}$
11     $\mathcal{P}_{T_i} \leftarrow$ actions induced by $V^c_{T_i,\mathcal{T}_{safe}}, \forall i \in I_t$
12     $(V, E) \leftarrow$taskGraph($\mathcal{T}^{ss}_{resp}, F$)
13     $\mathcal{P}_{T_{opt}}, I_{t_{opt}} \leftarrow$optTask($(V, E), \mathcal{P}_T, c, c_{tasks}$)
14     return ( $\mathcal{P}_W$, $\mathcal{P}_{T_{opt}}$, $I_{t_{opt}}$ )
```

The winning set and the feasible regions of each tasks are computed on the possible transitions encoded in $\mathcal{T}^{ss}_{resp}$ in line 6 of *alg.* (1). These are computed to be favourable states which satisfy task and steady state propositions and other imposed constraints since this is operated on a transition system which is filtered and pruned earlier. This is followed by reachability computation of states that can lead to the winning set in line 7 of *alg.* (1). In line 8 of *alg.* (1), intersection test of the initial state and the states that can reach the winning set determines existence of a feasible policy.

If feasible, then there exists a policy to reach the winning set from the initial state and policies within the winning set to reach all of the tasks infinitely often while complying all other propositional constraints. Further policy optimization is performed for recurring task ordering based on computed feasible policies and this is evaluated through a defined cost function.

2.3.2 Feasibility and Winning Set Computation

Referenced in line 6 of *alg.* (1) on the previous page, this checks feasibility of satisiying propositions that are specified for tasks and compute a winning set, which contains a set of preferrable states to be in in order to satisfy all tasks (eg: be able to endlessly visit all task regions in state space) while obeying all other specfied constraints which is encoded in the transition system, $\mathcal{T}^{ss}_{resp}$.

This module acts as a non-deterministic automaton checker for the task region under other specified constraints and also avoids exponential blowup of processing full LTL language and instead focus on the proposed LTL fragment. For non-deterministic systems, the increased branching factor of transitions causes another exponential factor in complexity relative to non-deterministic systems, thus contributing to doubly-exponential complexity using automaton approach over full LTL.

2.3.3 Value Computation With Respect to Reach Set

Referenced in (3) and (4), this is used in several different places to compute optimality in order to reach certain destinations, such as the winning set and individual task state space regions. Using a variant of graph search algorithm altered for non-determinism as evaluated by minimax cost function, we obtain feasibility in the worst case as encoded by the relevant transition system.

2.3.4 Task Graph Construction

Referenced in line 12 of *alg.* (1) on the preceding page, this is constructed from a subset of the entire transition system to represent different states related to task propositions. Different task optimizers can evaluate traversal cost calculation on this graph which encode feasible transitions to tasks while satisfying all other constraints.

2.3.5 Task Policy Optimizers

Referenced in line 13 of *alg.* (1) on the previous page, different task optimizer may be substituted for different purposes. We briefly mention a few suitable task cost optimizers below.

Minimax/Bottleneck optimizer considers the worst case scenario between a pair of task nodes. The complexity is shown to be very reasonable. The authors propose an implementation with a bounding search technique using median edge length between tasks as the bounding value. Average cost optimizer considers the average cost over discrete time steps. Average cost per task cycle optimizer considers cost over a limiting cycle of tasks as time tends to infinity and this proved to be as hard as the travelling sales man problem by the authors, but approximations can be used to decrease computation time.

3 Implementation and Demonstration

3.1 Sample Problem Formulation

Demonstration of use of the proposed solution by the authors is done by implementing the mentioned algorithms for synthesizing policies tailored to a specific setting as follows. An agent is controllable but a moving obstruction may not be controlled on a 2D grid. Furthermore, the moving obstruction introduces non-determinism by selecting an arbitrary possible action. At each time step, an action (moving 1 unit north/south/west/east) is taken by the agent and the moving obstruction simultaneously. Constraints are imposed using LTL fragment such as not allowing the agent to come in a close vicinity (1 unit Manhattan distance or closer) of the moving obstruction and tasks of visiting predefined locations infinitely often while staying within a selected region of 2D grid after some arbitrary point of time. In sample problems, unit cost of transitioning to adjacent positions in the 2D grid is assumed. The synthesized policy is finally simulated by the agent and the moving obstruction selects a possible action in a uniform probability. Checks are performed during the simulation at each time step to see if the agent violates LTL propositions.

The state space for the problem is a product space of 2D coordinates of the agent and the moving obstruction. A transition consists of the agent selecting an available action deterministically while the moving obstruction simultaneously selects a possible action non-deterministically. As a whole, each action of agent non-deterministically selects one of multiple possible transitions of the system. The synthesized policies assume that the full state of the system is observable and bases feasible actions from present state of the system.

3.2 Implementation Overview

To better understand the proposed framework by the authors, the proposed algorithms for feasible policy synthesis are implemented in a high level programming language. However, optimal policy synthesis for task set ordering is currently not covered in the implementation. Controller synthesis and system simulation are performed for different sample scenarios. Code implementation is done in the Rust language and can be found at `https://github.com/clearlycloudy/ltl_fragment_planning`.

Although details of construction of transition system (line 2 to 5 of *alg.* (1) on page 3)is not mentioned by the authors, a natural approach of creating transition system using directed graphs encoding state and action spaces is followed. Multiple sets of transition systems are constructed that are parameterized by different proposition constraints as indicated by the algorithms proposed by the authors. The implementation for these transition systems are described below.

In line 2 of *alg.* (1) on page 3, the original transition system is pruned of all transitions which does not satisfy φ_{resp}. The original transition system itself is constructed from the underlying mechanics of an actual system such as reachability of neighbouring states in the state space. In the sample problem formulation, the original transition system is constructed from a product space of agent coordinates and moving obstruction coordinates in a finite sized 2D grid. Then, the allowable transitions are all combinations of states where the agent's next coordinate is 1 unit away and the moving obstacle's next coordinate is 1 unit away or stationary, while constrained by grid dimensions. The moving obstacle's action is formulated to also include stationarity for debugging purposes in the implementation. Since $\varphi_{resp} := \bigwedge_{j \in I_r} \Box(\psi_{r,j} \implies \bigcirc \phi_{r,j})$, we prune any transition starting at $[[\psi_r]]$ that does not transition to states $[[\phi_r]]$. In (10) of Scenario 3, assuming a state space representation $(x_{agent}, y_{agent}, x_{obs}, y_{obs})$, transitions are pruned for (3,0,_,_) $\rightarrow$ (2,0,_,_), (3,0,_,_) $\rightarrow$ (4,0,_,_), (3,1,_,_) $\rightarrow$ (2,1,_,_), (3,1,_,_) $\rightarrow$ (4,1,_,_), (3,1,_,_) $\rightarrow$ (3,0,_,_), where "_" denotes any number within the grid dimension. The pruned transition system becomes $\mathcal{T}_{resp}$.

In line 3 of *alg.* (1) on page 3, φ_{resp} is pruned of a set of states that can definitely reach a set of states that violate φ_{safe} under worst-case conditions of non-determinism. This is done by considering the set which is not safe and computing its forced predecessor set (7), which is a set of state that can reach the unsafe set regardless of what the agent actions are at each state of a run if the run starts at a state in the predecessor set. The implementation of forced predecessor set uses the forced value function which performs a graph search and only assigns a number less than ∞ to states that are guaranteed to reach unsafe states. Once the forced predecessor set to unsafe states is computed, the part of the state space that does not contain this forced predecessor set is retained. Pruning is done on $\mathcal{T}_{resp}$ for bad transitions which has destination in the forced predecessor set to unsafe states as well as any related transitions in relevant starting state that share the same agent action as in the bad transitions. This extra condition is for guarantee in worst case scenario introduced by uncertainty of the system (in our scenario, the moving obstacle). The pruned transition system becomes $\mathcal{T}_{safe}$.

In line 4 of *alg.* (1) on page 3, exact same reasoning applied to obtain $\mathcal{T}_{safe}$ is applied here, but for φ_{per} which is a steady state version of φ_{safe}. The pruned transition system becomes $\mathcal{T}_{per}$.

In line 5 of *alg.* (1) on page 3, the same reasoning applied to obtain $\mathcal{T}_{resp}$ is applied, but targeted for φ^{ss}_{resp} which is a steady state version of φ_{resp}. The pruned transition system becomes $\mathcal{T}^{ss}_{resp}$.

Each of the created transition systems and graph searching algorithm for reachability computation is operated on top of previously created transition systems. Thus, this series of filtering narrows down the feasibility space of the problem.

In line 6 of *alg.* (1) on page 3, feasibility of the task proposition is computed on $\mathcal{T}^{ss}_{resp}$ which satisfy rest of LTL constraints. The exact algorithm can be found in the authors' paper [2]. Essentially, if there exists some states for each task set that can be reached from some states from each of all other task sets under some policy, then there exists feasible policy for the task propositions. This is achieved by reachability computation using the controlled predecessor set (6) and controlled value function (3) where the destination set is parameterized for each task set. Results of these are tested for set intersections to determine each task set reachability. Generally, multiple states exist for each task set due to product space and dimensionality explosion for complex systems. Finally the algorithm returns the favourable predecessor set, called the winning set (5) that are able to traverse to states that satisfy task sets infinitely often.

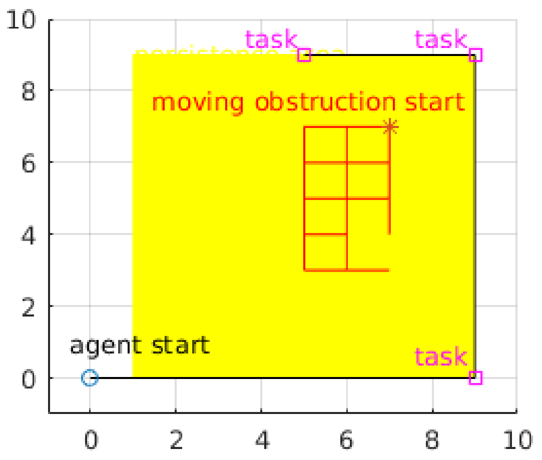
In line 10 of *alg.* (1) on page 3, feasible policies to reach the winning set from the initial condition of the system is computed by a graph search implementation of the value function. In line 11 of *alg.* (1) on page 3, policies to reach feasible subset of each task set are computed. Thus, the controller operates in 2 general regions, one for the transient phase and one for the steady state phase where tasks propositions are met. Using these, we can optimize for task set ordering using a chosen minimization function but it is not investigated further. In the end, the synthesis procedure produces a set of lookups for feasible policy given a present state of the entire system and the region of operation, as well as a specific ordering of task policies to cycle through in steady state.

3.3 Feasible Policy Synthesis and Scenario 1

A sample scenario of an agent repeatedly visiting task positions while avoiding coming within 1 unit distance of a moving obstacle is specified using the proposed LTL subset below.

$$\begin{aligned}
&\varphi = \varphi_{safe} \wedge \varphi_{resp} \wedge \varphi_{resp}^{ss} \wedge \varphi_{per} \wedge \varphi_{task} \qquad (8)\\
&\varphi_{safe} := \mathit{in\_grid} \wedge \neg \mathit{collide} \wedge \mathit{mov\_obs\_range}\\
&\varphi_{resp} := \{\emptyset\}\\
&\varphi_{resp}^{ss} := \{\emptyset\}\\
&\varphi_{per} := \Diamond\Box(\mathit{agent\ within}\ ((1,0),(9,9)))\\
&\varphi_{task} := \Box\Diamond(\mathit{agent\ visit}\ (5,9))\\
&\qquad \wedge \Box\Diamond(\mathit{agent\ visit}\ (9,9))\\
&\qquad \wedge \Box\Diamond(\mathit{agent\ visit}\ (0,9))\\
&\mathit{in\_grid} := \mathit{pos}_{agent}\ \mathit{within}((0,0),(9,9))\\
&\qquad \wedge \mathit{pos}_{mov\_obs}\ \mathit{within}((0,0),(9,9))\\
&\mathit{collide} := ||\mathit{pos}_{agent} - \mathit{pos}_{mov\_obs}||_1 \leq 1\\
&\mathit{mov\_obs\_range} := \mathit{pos}_{mov\_obs}\ \mathit{within}((5,3),(7,7))
\end{aligned}$$

A feasible policy synthesis is then performed using our constructed algorithm. At this point, the ordering of tasks are taken arbitrary and hence the policy is synthsized for feasibility only. This is followed by a simulation with the moving obstruction having uniformly distributed selection of possible actions and the agent controlled by the feasible policy constructed earlier. A sample run trajectory lasting 3 cycles of task set completions for the above scenario (8) is collected and post-processed below.



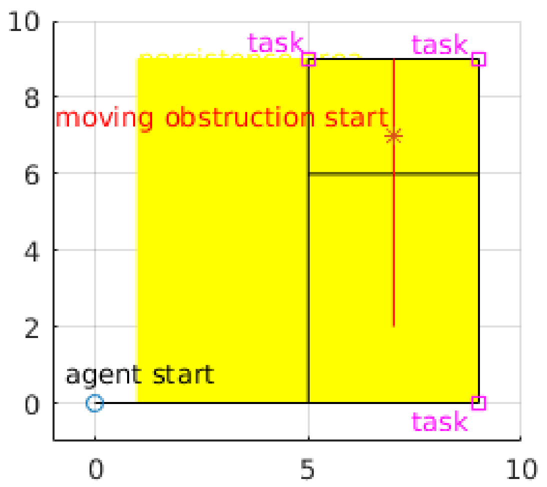
The synthesized feasible policy enables the agent to successfully pick actions that meets the sepcification.

3.4 Feasible Policy Synthesis and Scenario 2

The LTL specification is altered for the moving obstruction range in the previous scenario to see the difference in synthesized feasiblity policy.

$$
\begin{aligned}
\varphi &= \varphi_{safe} \land \varphi_{resp} \land \varphi^{ss}_{resp} \land \varphi_{per} \land \varphi_{task} \qquad (9)\\
\varphi_{safe} &:= in\_grid \land \neg collide \land mov\_obs\_range\\
\varphi_{resp} &:= \{\emptyset\}\\
\varphi^{ss}_{resp} &:= \{\emptyset\}\\
\varphi_{per} &:= \Diamond\Box(agent\ within\ ((1,0),(9,9)))\\
\varphi_{task} &:= \Box\Diamond(agent\ visit\ (5,9))\\
&\quad \land \Box\Diamond(agent\ visit\ (9,9))\\
&\quad \land \Box\Diamond(agent\ visit\ (0,9))\\
in\_grid &:= pos_{agent}\ within((0,0),(9,9))\\
&\quad \land pos_{mov\_obs}\ within((0,0),(9,9))\\
collide &:= ||pos_{agent} - pos_{mov\_obs}||_1 \leq 1\\
mov\_obs\_range &:= pos_{mov\_obs}\ within((7,2),(7,9))
\end{aligned}
$$

A sample run trajectory lasting 3 cycles of task set completions for the above scenario (9) is collected and post-processed below.



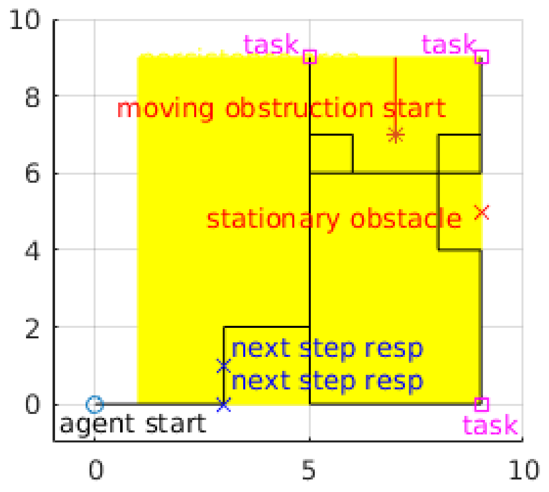
Using the generated policy that is constructed from the altered specification, the agent takes different actions based on the location of the moving obstruction. It is able to cross region of moving obstruction at some instances of time steps.

3.5 Feasible Policy Synthesis and Scenario 3

The LTL specification is altered by adjusting moving obstruction range and including stationary obstruction and next step response as follows.

$$
\begin{aligned}
\varphi &= \varphi_{safe} \land \varphi_{resp} \land \varphi^{ss}_{resp} \land \varphi_{per} \land \varphi_{task} \qquad (10)\\
\varphi_{safe} &:= in\_grid \land \neg collide_{mov} \land \neg collide_{stationary}\\
&\quad \land mov\_obs\_range\\
\varphi_{resp} &:= \Box(pos_{agent} = (3,0) \implies \bigcirc pos_{agent} = (3,1))\\
&\quad \land \Box(pos_{agent} = (3,1) \implies \bigcirc pos_{agent} = (3,2))\\
\varphi^{ss}_{resp} &:= \{\emptyset\}\\
\varphi_{per} &:= \Diamond\Box(agent\ within\ ((1,0),(9,9)))\\
\varphi_{task} &:= \Box\Diamond(agent\ visit\ (5,9))\\
&\quad \land \Box\Diamond(agent\ visit\ (9,9))\\
&\quad \land \Box\Diamond(agent\ visit\ (0,9))\\
in\_grid &:= pos_{agent}\ within((0,0),(9,9))\\
&\quad \land pos_{mov\_obs}\ within((0,0),(9,9))\\
collide_{mov} &:= ||pos_{agent} - pos_{mov\_obs}||_1 \leq 1\\
collide_{stationary} &:= ||pos_{agent} - pos_{stationary\_obs}||_1 = 0\\
mov\_obs\_range &:= pos_{mov\_obs}\ within((7,7),(7,9))
\end{aligned}
$$

A sample run trajectory lasting 3 cycles of task set completions for the above scenario (10) is collected and post-processed below.



3.6 Infeasible Policy Synthesis and Scenario 4

The previous sceneario is altered to test infeasibility by giving LTL specification which cannot be met. Specifically, the moving obstruction range is changed from ((7,7),(7,9)) to ((7,6),(7,9)). This change blocks the agent from reaching a task at (9,9) from the remaining tasks at (5,9) and (0,9) since the agent has to avoid stationary obstruction and not come within 1 unit distance of the moving obstruction.

$$\varphi = \varphi_{safe} \wedge \varphi_{resp} \wedge \varphi^{ss}_{resp} \wedge \varphi_{per} \wedge \varphi_{task} \tag{11}$$

$$\begin{aligned}
\varphi_{safe} &:= in\_grid \wedge \neg collide_{mov} \wedge \neg collide_{stationary} \\
&\quad \wedge mov\_obs\_range \\
\varphi_{resp} &:= \Box(pos_{agent} = (3,0) \implies \bigcirc pos_{agent} = (3,1)) \\
&\quad \wedge \Box(pos_{agent} = (3,1) \implies \bigcirc pos_{agent} = (3,2)) \\
\varphi^{ss}_{resp} &:= \{\emptyset\} \\
\varphi_{per} &:= \Diamond\Box(agent\ within\ ((1,0),(9,9))) \\
\varphi_{task} &:= \Box\Diamond(agent\ visit\ (5,9)) \\
&\quad \wedge \Box\Diamond(agent\ visit\ (9,9)) \\
&\quad \wedge \Box\Diamond(agent\ visit\ (0,9)) \\
in\_grid &:= pos_{agent}\ within((0,0),(9,9)) \\
&\qquad \wedge pos_{mov\_obs}\ within((0,0),(9,9)) \\
collide_{mov} &:= ||pos_{agent} - pos_{mov\_obs}||_1 \leq 1 \\
collide_{stationary} &:= ||pos_{agent} - pos_{stationary\_obs}||_1 = 0 \\
mov\_obs\_range &:= pos_{mov\_obs}\ within((7,6),(7,9))
\end{aligned}$$

As expected, feasibility synthesis procedure terminates with the conclusion of infeasible policy synthesis for the above scenario (11).

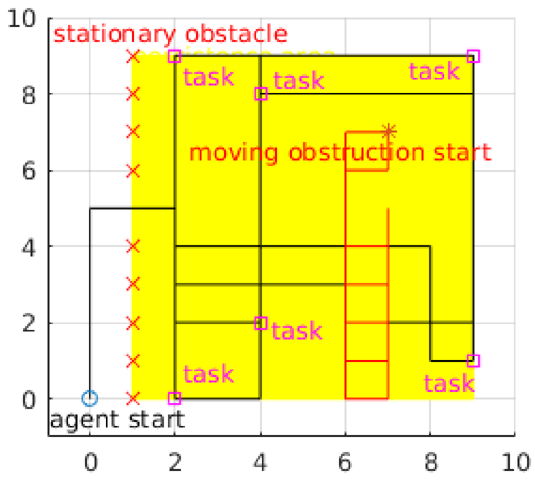
3.7 Feasible Policy Synthesis and Scenario 5

In this scenario, more tasks and staionary obstacles are included.

$$\varphi = \varphi_{safe} \wedge \varphi_{resp} \wedge \varphi^{ss}_{resp} \wedge \varphi_{per} \wedge \varphi_{task} \tag{12}$$

$$\begin{aligned}
\varphi_{safe} &:= in\_grid \wedge \neg collide_{mov} \wedge \neg collide_{stationary} \\
&\quad \wedge mov\_obs\_range \\
\varphi_{resp} &:= \{\emptyset\} \\
\varphi^{ss}_{resp} &:= \{\emptyset\} \\
\varphi_{per} &:= \Diamond\Box(agent\ within\ ((1,0),(9,9))) \\
\varphi_{task} &:= \Box\Diamond(agent\ visit\ (9,9)) \\
&\quad \wedge \Box\Diamond(agent\ visit\ (9,1)) \\
&\quad \wedge \Box\Diamond(agent\ visit\ (4,8)) \\
&\quad \wedge \Box\Diamond(agent\ visit\ (4,2)) \\
&\quad \wedge \Box\Diamond(agent\ visit\ (2,9)) \\
&\quad \wedge \Box\Diamond(agent\ visit\ (2,0)) \\
in\_grid &:= pos_{agent}\ within((0,0),(9,9)) \\
&\qquad \wedge pos_{mov\_obs}\ within((0,0),(9,9)) \\
collide_{mov} &:= ||pos_{agent} - pos_{mov\_obs}||_1 \leq 1 \\
collide_{stationary} &:= ||pos_{agent} - pos_{stationary\_obs_i}||_1 = 0 \\
&\qquad \forall stationary\_obs\ at\{(1,0),(1,1),(1,2), \\
&\qquad (1,3),(1,4),(1,6),(1,7),(1,8),(1,9)\} \\
mov\_obs\_range &:= pos_{mov\_obs}\ within((6,0),(7,7))
\end{aligned}$$

A sample run trajectory lasting 3 cycles of task set completions for the above scenario (12) is collected and post-processed below.



A feasible policy is generated in this case and the agent is seen to cross region of moving obstruction.

3.8 Infeasible Policy Synthesis and Scenario 6

In this scenario, LTL specification is altered from the previous scenario in the moving obstruction range from ((6,0),(7,7)) to ((6,0),(7,9)).

$$\varphi = \varphi_{safe} \wedge \varphi_{resp} \wedge \varphi_{resp}^{ss} \wedge \varphi_{per} \wedge \varphi_{task} \tag{13}$$

$$\begin{aligned}
\varphi_{safe} &:= in\_grid \wedge \neg collide_{mov} \wedge \neg collide_{stationary} \\
&\quad \wedge mov\_obs\_range \\
\varphi_{resp} &:= \{\emptyset\} \\
\varphi_{resp}^{ss} &:= \{\emptyset\} \\
\varphi_{per} &:= \Diamond\Box(agent\ within\ ((1,0),(9,9))) \\
\varphi_{task} &:= \Box\Diamond(agent\ visit\ (9,9)) \\
&\quad \wedge \Box\Diamond(agent\ visit\ (9,1)) \\
&\quad \wedge \Box\Diamond(agent\ visit\ (4,8)) \\
&\quad \wedge \Box\Diamond(agent\ visit\ (4,2)) \\
&\quad \wedge \Box\Diamond(agent\ visit\ (2,9)) \\
&\quad \wedge \Box\Diamond(agent\ visit\ (2,0)) \\
in\_grid &:= pos_{agent}\ within((0,0),(9,9)) \\
&\quad \wedge pos_{mov\_obs}\ within((0,0),(9,9)) \\
collide_{mov} &:= ||pos_{agent} - pos_{mov\_obs}||_1 \leq 1 \\
collide_{stationary} &:= ||pos_{agent} - pos_{stationary\_obs_i}||_1 = 0 \\
&\quad \forall stationary\_obs\ at\{(1,0),(1,1), \\
&\quad (1,2),(1,3),(1,4),(1,6),(1,7),(1,8), \\
&\quad (1,9)\} \\
mov\_obs\_range &:= pos_{mov\_obs}\ within((6,0),(7,9))
\end{aligned}$$

A policy cannot be synthesized as expected.

3.9 Statistics Related to Feasibility Synthesis

As analyzed by the authors, the complexity of proposed feasibility synthesis has a relatively good complexity when compared to full automaton approach. However, large state space dimensionality issue still exists. For each additional dimension of state space, the required complexity for storage increases exponentially. In the problem of one agent and one moving obstruction on a N-dimensional grid of size d in each dimension, there exists a maximal combination of d^{2N} states where the factor of 2 in the exponent comes from product space of agent and moving obstruction positions. For sample scenarios of an agent and moving obstruction on a 2D 10×10 grid, $d^{2N} = 10000$. As imaginable, inclusion of more agents and nondeterminism contributes to the complexity of storage by an exponential factor. This also adversely affect computational complexity of the proposed framework since it still depends on size of transition system which is exponentially related to the addition of more states. This limitation suggests significant optimization is needed for online or large state space problems.

The mentioned scenario experiments are done on a typical consumer laptop (Intel i3-6100U CPU @ 2.30GHz with 4GB of RAM). Time duration for controller synthesis is on the order of a few to a dozen seconds for the presented problem of an agent and moving obstruction on a 2D 10×10 grid. Some recorded statistics for the chosen scenarios are tabulated with respect to feasibility synthesis time, winning set size, and number of tasks.

Scenario	Synthesis Time	$\|W\|$	Tasks
1	∼13s	3025	3
2	∼8s	2461	3
3	∼5.2s	986	3
4	∼3s	0	3
5	∼ 19s	2982	6
6	∼ 16s	227	6

A typical number of elements in the winning set as computed by the feasibility algorithm is on the order of ∼1000. In scenarios with higher propositional constraints, a much smaller subset of the state space is determined to be feasible for the winning set and subsequent task policy synthesis is accelerated with reduced complexity due to decrease in size of winning set.

Futher optimization for computation and execution time such as data sparsity and multi-threaded approaches may help but they are not considered for the current implementation. Some of the operations for obtaining policy actions (measured to be lasting $\sim$2s for scenario 2) may be combined with other operations during the computation of the winning set, but is not done for clearity during implementation. Several sources of computation complexity are external to the main feasibility computation but rather preprocessing steps for state space permuation generation and filtering required for transition system construction.

4 Summary

A subset of LTL is investigated for feasible and optimal policy synthesis. Sample implementation and simulations which demonstrate the framework and algorithms are shown to be approachable for problems with small state space dimension. Generalization to high dimensional state space would require drastic optimization for online control synthesis applications. Uncertainty is investigated through usage of non-deterministic moving obstruction in shown scenarios. In this framework, exact numeric uncertainty value is not considered and instead the existence of uncertainty justifies synthesis procedure based on the worst-case scenario. Furthermore, this framework is extensible for introducing non-determinism for the agent's action as well which can be used to model some level of controller uncertainty and robustness. In particular, markov decision processes and monte carlo methods are used for determining robust policies in [3] and this is an area for future investigation.

References

[1] E.M. Wolff, U. Topcu, and R.M. Murray. "Efficient Reactive Controller Synthesis for a Fragment of Linear Temporal Logic". In: *Proc. of Int. Conf. on Robotics and Automation* (2012).

[2] E.M. Wolff, U. Topcu, and R.M. Murray. "Optimal Control of Non-deterministic Systems for a Computationally Efficient Fragment of Temporal Logic". In: *Conference on Decision and Control* (2013).

[3] E.M. Wolff, U. Topcu, and R.M. Murray. "Robust Control of Uncertain Markov Decision Processes with Temporal Logic Specifications". In: *Conference on Decision and Control* (2012).