Abstract

An obstacle that prevents the wide adoption of (deep) reinforcement learning (RL) in control systems is its need for a large amount of interactions with the environment in order to master a skill. The learned skill usually generalizes poorly across domains and re-training is often necessary when presented with a new task. We present a framework that combines methods in temporal logic (TL) with hierarchical reinforcement learning (HRL). The set of techniques we provide allows for convenient specification of tasks with complex logic, learn hierarchical policies (meta-controller and low-level controllers) with well-defined intrinsic rewards using any RL methods and is able to construct new skills from existing ones without additional learning. We evaluate the proposed methods in a simple grid world simulation as well as simulation on a Baxter robot.

1 Introduction

Reinforcement learning has received much attention in the recent years because of its achievements in games\cite{mnih2015human}, robotics \cite{jang2016deep},\cite{levine2016end},\cite{gu2016deep} and autonomous driving \cite{isele2017deep},\cite{madrigal2017end}. However, training a policy that sufficiently masters a skill requires an enormous amount of interactions with the environment and acquiring such experience can be difficult on physical systems. Moreover, most learned policies are tailored to mastering one skill (by maximizing the reward) and is hardly reusable on a new skill.

Skill composition is the idea of constructing new skills with existing skills (and hence their policies) with little to no additional learning. In stochastic optimal control, this idea has been adopted by authors of \cite{todorov2009general} and \cite{da2009skill} to construct provably optimal control laws based on linearly solvable Markov decision processes. Authors of \cite{haarnoja2017soft}, \cite{tang2017scalable} have showed in simulated manipulation tasks that approximately optimal policies can result from adding the Q-functions of the existing policies.

Hierarchical reinforcement learning is an effective means of achieving transfer among tasks. The goal is to obtain task-invariant low-level policies, and by re-training the meta-policy that schedules over the low-level policies, different skills can be obtain with less samples than training from scratch. Authors of \cite{heess2016learning} have adopted this idea in learning locomotor controllers and have shown successful transfer among simulated locomotion tasks. Authors of \cite{oh2017natural} have utilized a deep hierarchical architecture for multi-task learning using natural language instructions.

Temporal logic is a formal language commonly used in software and digital circuit verification\cite{baer2008temporal} as well as formal synthesis\cite{belta2017temporal}. It allows for convenient expression of complex behaviors and causal relationships. TL has been used by \cite{sadri2015synthesis},\cite{leahy2015automatic} to synthesis provably correct control policies. Authors of \cite{aksaray2016automatic} have also combined TL with Q-learning to learn satisfiable policies in discrete state and action spaces.

In this work, we focus on hierarchical skill acquisition and zero-shot skill composition. Once a set of skills are acquired, we provide a technique that can synthesize new skills without the need to further interact with the environment (given the state and action spaces as well as the transition remain the same). We adopt temporal logic as the task specification language. Compared to most
heuristic reward structures used in the RL literature to specify tasks, formal specification language excels at its semantic rigor and interpretability of specified behaviors. Our main contributions are:

- We take advantage of the transformation between TL formula and finite state automatons (FSA) to construct deterministic meta-controllers directly from the task specification without the necessity for additional learning. We show that by adding one discrete dimension to the original state space, structurally simple parameterized policies such as feed-forward neural networks can be used to learn tasks that require complex temporal reasoning.

- Intrinsic motivation has been shown to help RL agents learn complicated behaviors with less interactions with the environment [Singh et al. (2004) Kulkarni et al. (2016) Jaderberg et al. (2016)]. However, designing a well-behaved intrinsic reward that aligns with the extrinsic reward takes effort and experience. In our work, we construct intrinsic rewards directly from the input alphabets of the FSA (a component of the automaton), which guarantees that maximizing each intrinsic reward makes positive progress towards satisfying the entire task specification. From a user’s perspective, the intrinsic rewards are constructed automatically from the TL formula.

- In our framework, each FSA represents a hierarchical policy with low-level controllers can be re-modulated to achieve different tasks. Skill composition is achieved by manipulating the FSA that results from their TL specifications in a deterministic fashion. Instead of interpolating/extrapolating among existing skills, we present a simple policy switching scheme based on graph manipulation of the FSA. Therefore, the compositional outcome is much more transparent. We introduce a method that allows learning of such hierarchical policies with any non-hierarchical RL algorithm. Compared with previous work on skill composition, we impose no constraints on the policy representation or the problem class.

2 Preliminaries

2.1 The Options Framework in Hierarchical Reinforcement Learning

In this section, we briefly introduce the options framework [Sutton et al. (1998)], especially the terminologies that we will inherit in later sections. We start with the definition of a Markov Decision Process.

**Definition 1.** An MDP is defined as a tuple $\mathcal{M} = (T, S, A, p(\cdot|\cdot), R(\cdot, \cdot, \cdot))$, where $T$ is the length of a fixed time horizon; $S \subseteq \mathbb{R}^n$ is the state space; $A \subseteq \mathbb{R}^m$ is the action space ($S$ and $A$ can also be discrete sets); $p : S \times A \times S \rightarrow [0, 1]$ is the transition function with $p(s'|s, a)$ being the conditional probability density of taking action $a \in A$ at state $s \in S$ and ending up in state $s' \in S$; $R : S \times A \times S \rightarrow \mathbb{R}$ is the reward function. The goal is to find a policy $\pi^* : S \rightarrow A$ (or $\pi^* : S \times A \rightarrow [0, 1]$ for stochastic policies) that maximizes the expected return, i.e.

$$\pi^* = \arg \max_{\pi} \mathbb{E}_{\tau}^\pi [R(\tau_T)]$$

(1)

where $\tau_T = (s_0, a_0, ..., s_T,)$ denotes the state-action trajectory from time 0 to $T$.

The options framework exploits temporal abstractions over the action space. An option is defined as a tuple $o = (I, \pi^o, \beta)$ where $I$ is the set of states that option $o$ can be initiated (here we let $I = S$ for all options), $\pi^o : S \rightarrow A$ is an options policy and $\beta : S \rightarrow [0, 1]$ is the termination condition for the option. In addition, there is a policy over options $\pi^h : S \rightarrow \mathcal{O}$ (where $\mathcal{O}$ is a set of available options) that schedules among options. At a given time step $t$, an option $o$ is chosen according to $\pi^h(s_t)$ and the options policy $\pi^o$ is followed until $\beta(s) > \text{threshold}$ at time $t + k$, and the next option is chosen by $\pi^h(s_{t+k})$. 

Under review as a conference paper at ICLR 2018
2.2 scTLTL And Automata

In this work, we consider tasks specified with Truncated Linear Temporal Logic (TLTL). We restrict the set of allowed operators to be

\[
\phi := \top \mid f(s) < c \mid \neg \phi \mid \phi \land \psi \mid \phi \lor \psi \mid \diamond \phi \mid \phi U \psi \mid \phi T \psi \mid \square \phi \mid \phi \Rightarrow \ψ
\]

where \( f(s) < c \) is a predicate, \( \neg \) (negation/not), \( \land \) (conjunction/and), and \( \lor \) (disjunction/or) are Boolean connectives, and \( \diamond \) (eventually), \( T \) (then), \( \square \) (next), are temporal operators. Implication is denoted by \( \Rightarrow \) (implication). Essentially we excluded the \( \mathit{Always} \) operator \( \square \) with reasons similar to [Kupferman & Vardi (2001)]. We refer to this restricted TLTL as syntactically co-safe TLTL (scTLTL). There exists a real-value function \( \rho(s_{0:T}, \phi) \) called robustness degree that measures the level of satisfaction of trajectory \( s_{0:T} \) with respect to \( \phi \). \( \rho(s_{0:T}, \phi) \geq 0 \) indicates that \( s_{0:T} \) satisfies \( \phi \) and vice versa. Due to space constraints, we refer readers to [Li et al. (2016)] for definitions of TLTL and robustness degree.

Any scTLTL formula can be translated into a finite state automata (FSA) with the following definition:

**Definition 2.** An FSA is defined as a tuple \( A_\phi = \{Q_\phi, \Psi_\phi, q_0, p_\phi(\cdot|\cdot), \mathcal{F}_\phi\} \), where \( Q_\phi \) is a set of automaton states; \( \Psi_\phi \) is an input alphabet, we denote \( \psi_{q_i,q_j} \in \Psi_\phi \) the predicate guarding the transition from \( q_i \) to \( q_j \) (as illustrated in Figure 1); \( q_0 \in Q_\phi \) is the initial state; \( p_\phi : Q_\phi \times Q_\phi \rightarrow [0,1] \) is a conditional probability defined as

\[
p_\phi(q_j|q_i) = \begin{cases} 1 & \psi_{q_i,q_j} \text{ is true} \\ 0 & \text{otherwise} \end{cases}
\]

With each MDP state \( s \), we can calculate the transition in automata states at \( s \) by

\[
p_\phi(q_j|q_i, s) = \begin{cases} 1 & \rho(s, \psi_{q_i,q_j}) > 0 \\ 0 & \text{otherwise} \end{cases}
\]

We abuse the notation \( p_\phi \) to represent both kinds of transitions when the context is clear. \( \mathcal{F}_\phi \) is a set of final automaton states.

The translation from TLTL formula FSA to can be done automatically with available packages like Lomap [Ulusoy (2017)].

**Example 1.** Figure 1 (left) illustrates the FSA resulting from formula \( \phi = \neg b U a \). In English, \( \phi \) entails during a run, \( b \) cannot be true until \( a \) is true and \( a \) needs to be true at least once. The FSA has four automaton states \( Q_\phi = \{q_0, q_f, \text{trap}\} \) with \( q_0 \) being the input(initial) state (here \( q_f \) serves to track the progress in satisfying \( \phi \)). The input alphabet is defined as the \( \Psi_\phi = \{-a \land \neg b, \neg a \land b, a \land \neg b, a \land b\} \). Shorthands are used in the figure, for example \( a = (a \land b) \lor (a \land \neg b) \). \( \Psi_\phi \) represents the power set of \( \{a, b\} \), i.e. \( \Psi_\phi = 2^{\{a,b\}} \). During execution, the FSA always starts from state \( q_0 \) and transitions according to Equation 2 or 3. The specification is satisfied when \( q_f \) is reached and violated when \( \text{trap} \) is reached. In this example, \( q_f \) is reached only when \( a \) becomes true before \( b \) becomes true.

3 Problem Formulation And Approach

We start with the following problem definition:

**Problem 1.** Given an MDP in Definition 1 with unknown transition dynamics \( p(s'|s,a) \) and a scTLTL specification \( \phi \) over state predicates (along with its FSA \( A_\phi \)) as in Definition 2 Find a policy \( \pi^*_\phi : S \rightarrow A \) such that

\[
\pi^*_\phi = \arg \max_{\pi_\phi} \mathbb{E}_{\pi_\phi}[\mathbb{I}(\rho(s_{0:T}, \phi) > 0)],
\]

where \( \mathbb{I}(\rho(s_{0:T}, \phi) > 0) \) is an indicator function with value \( 1 \) if \( \rho(s_{0:T}, \phi) > 0 \) and \( 0 \) otherwise.
Problem 1 defines a policy search problem where the trajectories resulting from following the optimal policy should satisfy the given scTLTL formula in expectation.

**Problem 2.** Given two scTLTL formulas $\phi_1$ and $\phi_2$ along with policy $\pi_{\phi_1}$ that satisfies $\phi_1$ and $\pi_{\phi_2}$ that satisfies $\phi_2$. Obtain a policy $\pi_\phi$ that satisfies $\phi = \phi_1 \land \phi_2$.

Problem 2 defines the problem of task composition. Given two policies each satisfying a scTLTL specification, the policy that satisfies the conjunction of the given specifications can be synthesized. Solving this problem is useful when we want to break a complex task into simple and manageable components, learn a policy that satisfies each component and “stitch” all the components together so that the original task is satisfied. It can also be the case that as the scope of the task grows with time, the original task specification is amended with new items. Instead of having to re-learn the task from scratch, we can only learn a policy that satisfies the new items and combine them with the old policy.

We propose to solve Problem 1 by constructing a product MDP from the given MDP and FSA that can be solved using any state-of-the-art RL algorithm. The idea of using product automaton for control synthesis has been adopted in various literature [Leahy et al., 2015; Chen et al., 2012]. However, the methods proposed in these work are restricted to discrete state and actions spaces. We extend this idea to continuous state-action spaces and show its applicability on real robotics systems.

For Problem 2, we propose a policy switching scheme that guarantees to satisfy the compositional task specification. The switching policy takes advantage of the characteristics of FSA and uses robustness comparison at each step for decision making. Under mild constraints, the same switching policy can also be used to achieve zero-shot task space transfer. Details will be provided in the following sections.

### 4 FSA AUGMENTED MDP

Problem 1 can be solved with any episode-based RL algorithm. However, these methods suffer from sparse feedback because a reward signal can only be obtained at the end of each episode. To address this problem as well as setting up ground for automata guided HRL, we introduce the FSA augmented MDP

**Definition 3.** An FSA augmented MDP corresponding to scTLTL formula $\phi$ is defined as $\mathcal{M}_\phi = (T, \tilde{S}, \tilde{A}, \tilde{p}(\cdot | \cdot), \tilde{R}(\cdot, \cdot))$ where $T$ is the length of a fixed time horizon, $\tilde{S} \subseteq S \times Q_\phi$, $A$ is the same as the original MDP, $\tilde{p}(\tilde{s}'|\tilde{s}, a)$ is the probability of transitioning to $\tilde{s}'$ given $\tilde{s}$ and $a$, in particular

\[
\tilde{p}(\tilde{s}'|\tilde{s}, a) = \begin{cases} 
p(s'|s, a) & p_\phi(q'|q) = 1 \\
0 & \text{otherwise.} \end{cases}
\]

Here $p_\phi$ is defined in Equation (3). $\tilde{R} : \tilde{S} \times \tilde{S} \rightarrow \mathbb{R}$ is the FSA augmented reward function, defined by

\[
\tilde{R}(\tilde{s}, \tilde{s}') = \mathbb{1}(\rho(s', D_\phi^q) > 0),
\]

where $\Omega_q$ is the set of automata states that are connected with $q$ through outgoing edges. $D_\phi^q = \bigvee_{q' \in \Omega_q} \psi_{q,q'}$ represents the disjunction of all predicates guarding the transitions that originate from $q$.

As a quick example to the notation $D_\phi^q$, consider the state $q_0$ in the FSA in Figure 1. $\Omega_{q_0} = \{\text{trap}, q_f\}$, $D_\phi^{q_0} = \psi_{q_0, \text{trap}} \lor \psi_{q_0, q_f} = b \lor a$. The goal is then to find a policy $\pi : S \rightarrow A$ that maximizes the expected sum of $\tilde{R}$ over the horizon $T$. The FSA augmented MDP can be constructed with any standard MDP and a scTLTL formula. And it can be solved with any off-the-shelf RL algorithm. By directly learning the flat policy $\pi$ we bypass the need to learn multiple options policy separately. After obtaining the optimal policy...
Figure 1: FSA constructed from $\phi = \neg b \cup a$. (right): Specification amendment example. $A_{\phi_1}$ is constructed from $\phi_1 = \diamond a \land \diamond b$. $A_{\phi_2}$ is constructed from $\phi_2 = \neg b \cup a$. $A_\phi$ is constructed from $\phi = \phi_1 \land \phi_2$. The automaton state pair in parenthesis denote the corresponding states from $Q_{\phi_1}$ and $Q_{\phi_2}$ that the product state is constructed from. $q_i^0$ denotes the initial state of $A_{\phi_i}$, $q_i,j$ denotes the $j^{th}$ state of $Q_{\phi_i}$.

The reward function in Equation (7) encourages the system to exit the current automata state and move on to the next, and by doing so eventually reach the final state $q_f$. However, this reward does not distinguish between the trap state and other states and therefore will also promote entering of the trap state. One way to address this issue is to impose a terminal reward on both $q_f$ and trap. Because the reward is an indicator function with maximum value of 1, we assign terminal rewards $R_{q_f} = 2$ and $R_{\text{trap}} = -2$.

Appendix D describes the typical learning routine using FSA augmented MDP. The algorithm utilizes memory replay which is popular among off-policy RL methods (DQN, A3C, etc) but this is not a requirement for learning with $M_\phi$. On-policy methods can also be used.

5 Automata Guided Task Composition

To provide a solution for Problem 2 we will start by constructing the FSA of $\phi$ from that of $\phi_1$ and $\phi_2$ with the following definition.

Definition 4. Given $A_{\phi_1} = \langle Q_{\phi_1}, \Psi_{\phi_1}, q_1^1, p_{\phi_1}, F_{\phi_1} \rangle$ and $A_{\phi_2} = \langle Q_{\phi_2}, \Psi_{\phi_2}, q_2^1, p_{\phi_2}, F_{\phi_2} \rangle$. The FSA of $\phi$ is the product automaton of $A_{\phi_1}$ and $A_{\phi_2}$, i.e. $A_{\phi = \phi_1 \land \phi_2} = A_{\phi_1} \times A_{\phi_2} = \langle Q_{\phi}, \Psi_{\phi}, q_0, p_{\phi}, F_{\phi} \rangle$ where $Q_{\phi} \subseteq Q_{\phi_1} \times Q_{\phi_2}$ is the set of product automaton states, $q_0 = (q_0^1, q_0^2)$ is the product initial state, $F \subseteq F_{\phi_1} \times F_{\phi_2}$ is the final accepting states. Following Definition 2 for states $q = (q_1, q_2) \in Q_{\phi}$ and $q' = (q'_1, q'_2) \in Q_{\phi}$, the transition probability $p_{\phi}$ is defined as

$$p_{\phi}(q'|q) = \begin{cases} 1 & p_{\phi_1}(q'_1|q_1)p_{\phi_2}(q'_2|q_2) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

(9)

Example 2. Figure 1 (right) illustrates the FSA of $A_{\phi_1}$ and $A_{\phi_2}$ and their product automaton $A_\phi$. Here $\phi_1 = \diamond a \land \diamond b$ which entails that both $a$ and $b$ needs to be true at least once (order does not matter), and $\phi_2 = \neg b \cup a$ which is the same as Example 1. The resultant product corresponds to the formula $\phi = (\diamond a \land \diamond b) \land (\neg b \cup a)$ which dictates that $a$ and $b$ need to be true at least once, and $a$ needs to be true before $b$ becomes true (an ordered visit). We can see that the trap state occurs in $A_{\phi_2}$ and $A_\phi$, this is because if $b$ is ever true before $a$ is true, the specification is violated and $q_f$ can never be reached. In the product automaton, we aggregate all state pairs with a trap state component into one trap state.
For \( q = (q_1, q_2) \in Q_\phi \), let \( \Psi_q, \Psi_{q_1} \) and \( \Psi_{q_2} \) denote the set of predicates guarding the outgoing edges of \( q, q_1 \) and \( q_2 \) respectively. Equation (9) entails that a transition at \( q \) in the product automaton \( A_\phi \) exists only if corresponding transitions at \( q_1, q_2 \) exist in \( A_{\phi_1} \) and \( A_{\phi_2} \) respectively. Therefore, \( \psi_{q,q'} = \psi_{q_1,q_1}' \land \psi_{q_2,q_2}' \), for \( \psi_{q,q'} \in \Psi_q, \psi_{q_1,q_1}' \in \Psi_{q_1}, \psi_{q_2,q_2}' \in \Psi_{q_2} \). Following Equation (8),

\[
\pi_q^* = \arg \max_{\pi_q} \mathbb{E}^{\pi_q} \left[ \sum_{t=0}^{T-1} I(\rho(s_{t+1}, D_\phi^q) > 0) \right],
\]

where \( D_\phi^q = \bigvee_{q_1,q_2} (\psi_{q_1,q_1}' \land \psi_{q_2,q_2}') \).

Repeatedly applying the distributive law \((\Delta \land \Omega_1) \lor (\Delta \land \Omega_2) = \Delta \land (\Omega_1 \lor \Omega_2)\) to the logic formula \( D_\phi^q \) transforms the formula to

\[
D_\phi^q = (\bigvee_{q_1} \psi_{q_1,q_1}') \land (\bigvee_{q_2} \psi_{q_2,q_2}') = D_{\phi_1}^{q_1} \land D_{\phi_2}^{q_2}. \quad (11)
\]

Therefore,

\[
\pi_q^* = \arg \max_{\pi_q} \mathbb{E}^{\pi_q} \left[ \sum_{t=0}^{T-1} I(\rho(s_{t+1}, D_{\phi_1}^{q_1} \land D_{\phi_2}^{q_2}) > 0) \right] \quad (12)
\]

Given flat policies \( \pi_{\phi_1}, \pi_{\phi_2} \) that satisfies \( \phi_1 \) and \( \phi_2 \), following Equation (8), optimal options policy corresponding to their automaton states are \( \pi_{q_1,i}, \pi_{q_2,i} \) \( (q_1,i \text{ denotes the } i^{th} \text{ state in } Q_{\phi_1}) \). We propose the following switching policy that satisfies \( \phi = \phi_1 \land \phi_2 \)

\[
\pi_\phi(s,q) = \begin{cases} 
\pi_{\phi_1}(s,q_1) & \rho(s_t,D_{\phi_1}^{q_1}) < \rho(s_t,D_{\phi_2}^{q_2}) \\
\pi_{\phi_2}(s,q_2) & \text{otherwise}
\end{cases} \quad (13)
\]

We show empirically the above switching policy for skill composition in the following sections.

## 6 Experiments and Discussion

### 6.1 Grid World Simulation

In this section, we provide a simple grid world navigation example to illustrate the techniques presented in Sections 4 and 5. Here we have a robot navigating in a discrete 1 dimensional space. Its MDP state space \( S = \{s|s \in [-5,5], s \text{ is discrete}\} \), its action space \( A = \{left, stay, right\} \). The robot navigates in the commanded direction with probability 0.8, and with probability 0.2 it randomly chooses to go in the opposite direction or stay in the same place. The robot stays in the same place if the action leads it to go out of bounds.

We define two regions \( a : -3 < s < -1 \) and \( b : 2 < s < 4 \). For the first task, the scTLTL specification \( \phi_1 = \Diamond a \land \Diamond b \) needs to be satisfied. In English, \( \phi_1 \) entails that the robot needs to visit regions \( a \) and \( b \) at least once. To learn a deterministic optimal policy \( \pi_\phi^* : S \times Q \rightarrow A \), we use standard Q-Learning [Watkins 1989] on the FSA augmented MDP for this problem. We used a learning rate of 0.1, a discount factor of 0.99, epsilon-greedy exploration strategy with \( \epsilon \) decaying linearly from 0.0 to 0.01 in 1500 steps. The episode horizon is \( T = 50 \) and trained for 500 iterations. All Q-values are initialized to zero. The resultant optimal policy is illustrated in Figure 5.

We can observe from the figure above that the policy on each automaton state \( q \) serves a specific purpose. \( \pi_{q_0}^* \) tries to reach region \( a \) or \( b \) depending on which is closer. \( \pi_{q_1}^* \) always proceeds to region
The states with stay as their action are either goal regions (states \((-2, q_0), (3, q_1), \) etc) where a transition on \(q\) happens or states that are never reached (states \((-3, q_1), (-4, q_2), \) etc) because a transition on \(q\) occurs before they can be reached.

To illustrate automata guided task composition described in Example 2 instead of learning the task described by \(\phi\) from scratch, we can simply learn policy \(\pi_{\phi_2}\) for the added requirement \(\phi_2 = \neg b \cup a\). We use the same learning setup and the resultant optimal policy is depicted in Figure 4. It can be observed that \(\pi_{\phi_2}\) tries to reach \(a\) while avoiding \(b\). This behavior agrees with the specification \(\phi_2\) and its FSA provided in Figure 2. The action at \(s = 4\) is stay because in order for the robot to reach \(a\) it has to pass through \(b\), therefore it prefers to obtain a low reward over violating the task.

Having learned policies \(\pi_{\phi_1}\) and \(\pi_{\phi_2}\), we can now use Equation 13 to construct policy \(\pi_{\phi_1 \wedge \phi_2}\). The resulting policy for \(\pi_{\phi_1 \wedge \phi_2}\) is illustrated in Figure 2 (upper right). This policy guides the robot to first reach \(a\) (except for state \(s = 4\)) and then go to \(b\) which agrees with the specification.

Looking at Figure 1, the FSA of \(\phi = \phi_1 \wedge \phi_2\) have two options policies \(\pi_{\phi_1}(:, q_0)\) and \(\pi_{\phi_2}(:, q_1)\) (trap state and \(q_i\) are terminal states which don’t have options). State \(q_1\) has only one outgoing edge with the guarding predicate \(\psi_{q_1, q_1} : b\), which means \(\pi_{\phi_1}(:, q_1) = \pi_{\phi_2}(:, q_2)\) (they have the same guarding predicate). Policy \(\pi_{\phi}(\cdot, q_0)\) is a switching policy between \(\pi_{\phi_1}(\cdot, q_0)\) and \(\pi_{\phi_2}(\cdot, q_0)\). Figure 2 (lower left) shows the robustness comparison at each state. The policy with lower robustness is chosen following Equation 13. We can see that the robustness of both policies are the same from \(s = -5\) to \(s = 0\). And their policies agree in this range (Figures 3 and 4). As \(s\) becomes larger, disagreement emerge because \(\pi_{\phi_1}(\cdot, q_0)\) wants to stay closer to \(b\) but \(\pi_{\phi_2}(\cdot, q_0)\) wants otherwise. To maximize the robustness of their conjunction, the decisions of \(\pi_{\phi_2}(\cdot, q_0)\) are chosen for states \(s > 0\).

6.2 Simulated Baxter

In this section, we construct a set of more complicated tasks that require temporal reasoning and evaluate the proposed techniques on a Baxter robot. The environment is shown in Figure 5 (left). In front of the robot are three square regions and two circular regions. An object with planar coordinates \(p = (x, y)\) can use predicates \(S_{\text{red}}(p), S_{\text{blue}}(p), S_{\text{black}}(p), C_{\text{red}}(p), C_{\text{blue}}(p)\) to evaluate whether or not it is within the regions. The predicates are defined by \(S : x_{\text{min}} < x < x_{\text{max}}, y_{\text{min}} < y < y_{\text{max}}\) and \(C : \text{dist}((x, y), (x, y)_{\text{center}}) < r\), where \((x, y)\) is the planar coordinate in the world frame,
We design seven tasks each specified by a scTLTL formula. The task specifications and their English translations are provided in Appendix A. Throughout the experiments in this section, we use proximal policy search [Schulman et al. (2017)] as the policy optimization method. The hyperparameters are kept fixed across the experiments and are listed in Appendix B. The policy is a Gaussian distribution parameterized by a feed-forward neural network with 2 hidden layers each of 64 relu units. The state and action spaces vary across tasks and comparison cases, and are described in Appendix C.

We use the first task \( \phi_1 \) to evaluate the learning outcome using the FSA augmented MDP. As comparisons, we design two other rewards structures. The first is to use the robustness \( \rho(s_0; T, \phi) \) as the terminal reward for each episode and zero everywhere else, the second is a heuristic reward that aims to aligns with \( \phi_1 \). The heuristic reward consists of a tracking state and a set of quadratic distance function. For \( \phi_1 \), the heuristic reward is

\[
    r_{\phi_1} = \begin{cases} 
        -\text{dist}(p_{\text{redbox}}, p_{\text{redsquare center}}) & \text{if } p_{\text{sphere}} \text{ is in red circle} \\
        -\text{dist}(p_{\text{redbox}}, p_{\text{black square center}}) & \text{otherwise.} 
    \end{cases}
\]  

Heuristic rewards for other tasks are defined in a similar manner and are not presented explicitly due to space constraints.

The results are illustrated in Figure 3 (right). The upper right plot shows the average robustness over training iterations. Robustness is chosen as the comparison metric for its semantic rigor (robustness greater than zero satisfies task the specification). The reported values are averaged over 60 episodes and the plot shows the mean and 2 standard deviations over 5 random seeds. From the plot we can observe that the FSA augmented MDP and the terminal robustness reward performed comparatively in terms of convergence rate, whereas the heuristic reward fails to learn the task. The FSA augmented MDP also learns a policy with lower variance in final performance.

We deploy the learned policy on the robot in simulation and record the task success rate. For each of the three cases, we deploy the 5 policies learned from 5 random seeds on the robot and perform 10 sets of tests with randomly initialized states resulting in 50 test trials for each case. The average success rate is presented in Figure 3 (lower right). From the results we can see that the FSA augmented MDP is able to achieve the highest rate of success and this advantage over the robustness reward is due to the low variance of its final policy.
To evaluate the policy switching technique for skill composition, we first learn four policies $\pi_{\phi_2}$, $\pi_{\phi_3}$, $\pi_{\phi_4}$, $\pi_{\phi_5}$ using the FSA augmented MDP. Then we construct $\pi_{\phi_6} = \pi_{\phi_2 \land \phi_3}$ and $\pi_{\phi_7} = \pi_{\phi_2 \land \phi_3 \land \phi_4 \land \phi_5}$ using Equation (13) (It is worth mentioning that the policies learned by the robustness and heuristic rewards do not have an automaton state in them, therefore the skill composition technique does not apply). We deploy $\pi_{\phi_6}$ and $\pi_{\phi_7}$ on tasks 6 and 7 for 10 trials and record the average robustness of the resulting trajectories. As comparisons, we also learn tasks 6 and 7 from scratch using terminal robustness rewards and heuristic rewards, the results are presented in Figure 4. We can observe from the graphs that the complexity of the tasks increase, using the robustness and heuristic rewards fail to learn a policy that satisfies the specifications while the constructed policy can reliably achieve a robustness of greater than zero. We perform the same deployment test as previously described and looking at Figure 4 (right) we can see that for both tasks 6 and 7, only the policies constructed by skill composition are able to consistently complete the tasks.

7 Conclusion

In this paper, we proposed the FSA augmented MDP, a product MDP that enables effective learning of hierarchical policies using any RL algorithm for tasks specified by scTLTL. We also introduced automata guided skill composition, a technique that combines existing skills to create new skills without additional learning. We show in robotic simulations that using the proposed methods we enable simple policies to perform logically complex tasks.

Limitations of the current framework include discontinuity at the point of switching (for Equation (13)), which makes this method suitable for high level decision tasks but not for low level control tasks. The technique only compares robustness at the current step and choose to follow a sub-policy for one time-step, this makes the switching policy short-sighted and may miss long term opportunities. One way to address this is to impose a termination condition for following each sub-policies and terminate only when the condition is triggered (as in the original options framework).

REFERENCES


Appendix

A Task Specifications And Heuristic Rewards

<table>
<thead>
<tr>
<th>Task</th>
<th>scTLTL Formula</th>
<th>English Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>φ₁</td>
<td>(C&lt;sub&gt;red&lt;/sub&gt;(p&lt;sub&gt;sphere&lt;/sub&gt;) → ♦S&lt;sub&gt;red&lt;/sub&gt;(p&lt;sub&gt;redbox&lt;/sub&gt;)) ∧ (¬C&lt;sub&gt;red&lt;/sub&gt;(p&lt;sub&gt;sphere&lt;/sub&gt;) → ♦S&lt;sub&gt;black&lt;/sub&gt;(p&lt;sub&gt;redbox&lt;/sub&gt;))</td>
<td>If sphere in red circle, then red box eventually in red square. Otherwise red box eventually in black square</td>
</tr>
<tr>
<td>φ₂</td>
<td>(C&lt;sub&gt;red&lt;/sub&gt;(p&lt;sub&gt;sphere&lt;/sub&gt;) → ♦S&lt;sub&gt;red&lt;/sub&gt;(p&lt;sub&gt;redbox&lt;/sub&gt;)) ∧ (¬(C&lt;sub&gt;red&lt;/sub&gt;(p&lt;sub&gt;sphere&lt;/sub&gt;) ∨ (S&lt;sub&gt;back&lt;/sub&gt;(p&lt;sub&gt;sphere&lt;/sub&gt;) → ♦S&lt;sub&gt;black&lt;/sub&gt;(p&lt;sub&gt;redbox&lt;/sub&gt;)))</td>
<td>If sphere is not in red circle or black square, then red box eventually in red square. If sphere in blue circle, then blue box eventually in blue square.</td>
</tr>
<tr>
<td>φ₃</td>
<td>(C&lt;sub&gt;blue&lt;/sub&gt;(p&lt;sub&gt;sphere&lt;/sub&gt;) → ♦S&lt;sub&gt;blue&lt;/sub&gt;(p&lt;sub&gt;bluebox&lt;/sub&gt;)) ∧ (¬(C&lt;sub&gt;red&lt;/sub&gt;(p&lt;sub&gt;sphere&lt;/sub&gt;) ∨ (S&lt;sub&gt;back&lt;/sub&gt;(p&lt;sub&gt;sphere&lt;/sub&gt;) → ♦S&lt;sub&gt;black&lt;/sub&gt;(p&lt;sub&gt;bluebox&lt;/sub&gt;)))</td>
<td>If sphere is not in blue circle or black square, then blue box eventually in black square</td>
</tr>
<tr>
<td>φ₄</td>
<td>S&lt;sub&gt;black&lt;/sub&gt;(p&lt;sub&gt;sphere&lt;/sub&gt;) → ♦S&lt;sub&gt;blue&lt;/sub&gt;(p&lt;sub&gt;redbox&lt;/sub&gt;)</td>
<td>If sphere in black square, then eventually red box in blue square</td>
</tr>
<tr>
<td>φ₅</td>
<td>S&lt;sub&gt;black&lt;/sub&gt;(p&lt;sub&gt;sphere&lt;/sub&gt;) → ♦S&lt;sub&gt;red&lt;/sub&gt;(p&lt;sub&gt;bluebox&lt;/sub&gt;)</td>
<td>If sphere in black square, then eventually blue box in red square</td>
</tr>
<tr>
<td>φ₆</td>
<td>φ₂ ∧ φ₃</td>
<td>Conjunction of tasks 2 and 3</td>
</tr>
<tr>
<td>φ₇</td>
<td>φ₂ ∧ φ₃ ∧ φ₄ ∧ φ₅</td>
<td>Conjunction of tasks 2, 3, 4, 5</td>
</tr>
</tbody>
</table>

B Hyperparameters For Proximal Policy Optimization

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. Hidden Layers</td>
<td>2</td>
</tr>
<tr>
<td>Num. Units per layer</td>
<td>64</td>
</tr>
<tr>
<td>Activation</td>
<td>Relu</td>
</tr>
<tr>
<td>Policy Learning Rate</td>
<td>0.009</td>
</tr>
<tr>
<td>Value Learning Rate</td>
<td>0.009</td>
</tr>
<tr>
<td>Discount</td>
<td>0.99</td>
</tr>
<tr>
<td>Batch Size</td>
<td>60</td>
</tr>
<tr>
<td>GAE parameter</td>
<td>0.99</td>
</tr>
<tr>
<td>Num. Iterations</td>
<td>100</td>
</tr>
<tr>
<td>Num. Epochs</td>
<td>20</td>
</tr>
<tr>
<td>Clipping Parameter ε</td>
<td>0.2</td>
</tr>
<tr>
<td>Horizon</td>
<td>20</td>
</tr>
</tbody>
</table>

C State And Action Spaces

<table>
<thead>
<tr>
<th>Task</th>
<th>State Space</th>
<th>Action Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>φ₁</td>
<td>s = (p&lt;sub&gt;sphere&lt;/sub&gt;, p&lt;sub&gt;redbox&lt;/sub&gt;)</td>
<td>a = (p&lt;sub&gt;redbox&lt;/sub&gt;)&lt;sub&gt;target&lt;/sub&gt;</td>
</tr>
<tr>
<td>φ₂</td>
<td>s = (p&lt;sub&gt;sphere&lt;/sub&gt;, p&lt;sub&gt;redbox&lt;/sub&gt;)</td>
<td>a = (p&lt;sub&gt;redbox&lt;/sub&gt;)&lt;sub&gt;target&lt;/sub&gt;</td>
</tr>
<tr>
<td>φ₃</td>
<td>s = (p&lt;sub&gt;sphere&lt;/sub&gt;, p&lt;sub&gt;bluebox&lt;/sub&gt;)</td>
<td>a = (p&lt;sub&gt;redbox&lt;/sub&gt;)&lt;sub&gt;target&lt;/sub&gt;</td>
</tr>
<tr>
<td>φ₄</td>
<td>s = (p&lt;sub&gt;sphere&lt;/sub&gt;, p&lt;sub&gt;redbox&lt;/sub&gt;)</td>
<td>a = (p&lt;sub&gt;redbox&lt;/sub&gt;)&lt;sub&gt;target&lt;/sub&gt;</td>
</tr>
<tr>
<td>φ₅</td>
<td>s = (p&lt;sub&gt;sphere&lt;/sub&gt;, p&lt;sub&gt;bluebox&lt;/sub&gt;)</td>
<td>a = (p&lt;sub&gt;redbox&lt;/sub&gt;)&lt;sub&gt;target&lt;/sub&gt;</td>
</tr>
<tr>
<td>φ₆</td>
<td>s = (p&lt;sub&gt;sphere&lt;/sub&gt;, p&lt;sub&gt;redbox&lt;/sub&gt;, p&lt;sub&gt;bluebox&lt;/sub&gt;)</td>
<td>a = (p&lt;sub&gt;redbox&lt;/sub&gt;)&lt;sub&gt;target&lt;/sub&gt;</td>
</tr>
<tr>
<td>φ₇</td>
<td>s = (p&lt;sub&gt;sphere&lt;/sub&gt;, p&lt;sub&gt;redbox&lt;/sub&gt;, p&lt;sub&gt;bluebox&lt;/sub&gt;)</td>
<td>a = (p&lt;sub&gt;redbox&lt;/sub&gt;)&lt;sub&gt;target&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
D Learning With FSA Augmented MDP (off-policy version)

Algorithm 1 Automata Guided RL (off-policy version)

1: **Inputs:** Episode horizon $T$, $\hat{M}_\phi$ (consisting of an MDP and FSA $A_\phi$), maximum size for replay pool $N$
2: Initialize parameterized policy $\pi^\theta$ $\triangleright \theta$ is the policy parameters
3: Initialize replay pool $\mathcal{B} \leftarrow \{\}$
4: for $n = 1$ to number of training episodes do
5:   Select initial state $\hat{s}_0 = (s_0, q_0)$ $\triangleright s_0$ can be randomly selected, $q_0$ is always the initial automaton state $q^\phi$
6:   for $t = 0$ to $T$ do
7:      $\hat{s}_{t+1} = \text{GetNextState}(\hat{s}_t, a_t)$
8:      if $q_{t+1} == q_f$ then
9:         $\tilde{r}_t = 2$ $\triangleright$ terminal reward for satisfying $\phi$
10:        break $\triangleright \phi$ is satisfied, restart episode
11:   else if $q_{t+1} == \text{trap}$ then
12:      $\tilde{r}_t = -2$ $\triangleright$ terminal reward for violating $\phi$
13:        break $\triangleright \phi$ is violated, restart episode
14:   else
15:      $\tilde{r}_t = \text{GetReward}(\hat{s}_t, \hat{s}_{t+1})$ $\triangleright$ using Equation [7]
16:   end if
17:   $\mathcal{B} \leftarrow (\hat{s}_t, a_t, \hat{s}_{t+1}, \tilde{r}_t)$ $\triangleright$ store experience in replay pool
18:   if size($\mathcal{B}$) $>$ $N$ then
19:      pop($\mathcal{B}[0]$)
20:   end if
21:   $\theta \leftarrow \text{UpdatePolicy}(\mathcal{B})$ $\triangleright$ this can be any RL update rule and doesn’t necessarily have to occur at this location
22: end for
23: end for