Skill Transfer for Temporally-Extended Task Specifications

Jason Xinyu Liu[*] **Ankit Shah**[*] **Eric Rosen, George Konidaris, Stefanie Tellex**
Department of Computer Science
Brown University
{xinyu_liu, ankit_j_shah, eric_rosen, george_konidaris, stefanie_tellex}
@brown.edu

Abstract

Deploying robots in real-world domains, such as households and flexible manufacturing lines, requires the robots to be taskable on demand. Linear temporal logic (LTL) is a widely-used specification language with a compositional grammar that naturally induces commonalities across tasks. However, the majority of prior research on reinforcement learning with LTL specifications treats every new formula independently. We propose LTL-Transfer, a novel algorithm that enables subpolicy reuse across tasks by segmenting policies for training tasks into portable transition-centric skills capable of satisfying a wide array of unseen LTL specifications while respecting safety-critical constraints. Our experiments in a Minecraft-inspired domain demonstrate the capability of LTL-Transfer to satisfy over 90% of 500 unseen tasks while training on only 50 task specifications and never violating a safety constraint. We also deployed LTL-Transfer on a quadruped mobile manipulator in a household environment to show its ability to transfer to many fetch and delivery tasks in a zero-shot fashion.

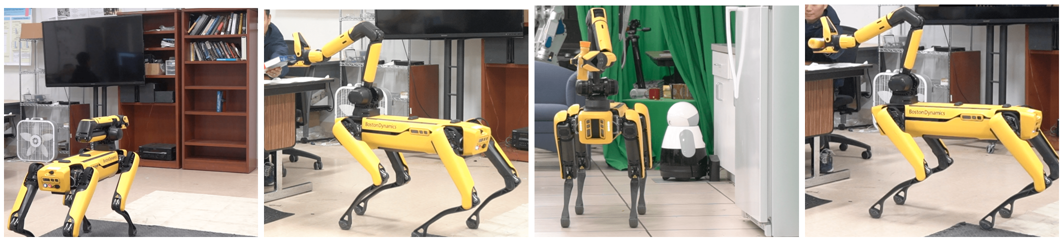

(a) pick up book (b) deliver book (c) pick up juice bottle (d) deliver juice bottle

Figure 1: The robot is executing 4 transition-centric options sequencially, each of which is transferred from a training task. They are composed together to solve a novel task, $\mathbf{F}(book \wedge \mathbf{F}(desk_a \wedge \mathbf{F}(juice \wedge \mathbf{F} desk_a)))$, i.e. fetch and deliver a book and a bottle of juice to the user.

1 Introduction

A key requirement for deploying autonomous agents in many real-world domains is the ability to perform multiple novel potential tasks on demand. These tasks typically share components like the objects and the trajectory segments involved, which creates the opportunity to reuse knowledge across tasks [24]. For example, a service robot on the factory floor might have to fetch the same set

[*]Equal contribution.

of components but in different orders depending on the product being assembled, in which case it should only need to learn to fetch a component once.

Linear temporal logic (LTL) [21] is becoming a popular means of specifying an objective for a reinforcement learning agent [17, 25, 6]. Its compositional grammar reflects the compositional nature of most tasks. However, most prior approaches to reinforcement learning for LTL specifications restart learning from scratch for each LTL formula. We propose LTL-Transfer, a novel algorithm that exploits the compositionality inherent to LTL task specifications to enable an agent to maximally reuse policies learned in prior LTL formulas to satisfy new, unseen specifications without additional training. For example, a robot that has learned to fetch a set of components on the factory floor should be able to fetch it in any order. LTL-Transfer also ensures that transferred subpolicies do not violate any safety constraints.

We demonstrated the efficacy of LTL-Transfer in a Minecraft-inspired domain, where the agent can complete over 90% of 500 new task specifications by training on only 50 specifications. Further, we demonstrate that it is possible to transfer satisfying policies with as few as 5 training specifications for certain classes of LTL formulas. We then deployed LTL-Transfer on a quadruped mobile manipulator to show its zero-shot transfer ability in a real-world household environment when performing with fetch and delivery tasks.

2 Preliminaries

Linear temporal logic (LTL) for task specification: LTL is a promising alternative to a numerical reward function as a means of expressing task specifications. An LTL formula φ is a boolean function that determines whether a given trajectory has satisfied the objective expressed by the formula. Littman et al. [17] argue that such task specifications are more natural than numerical reward functions, and they have subsequently been used as a target language for acquiring task specifications in several settings, including from natural language [20] and learning from demonstration [22]. Formally, an LTL formula is interpreted over traces of Boolean propositions over discrete time, and is defined through the following recursive syntax:

$$\varphi := p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathbf{X}\varphi \mid \varphi_1 \,\mathbf{U}\, \varphi_2 \tag{1}$$

Here p represents an atomic proposition, mapping a state to a boolean value; $\varphi,\ \varphi_1,\ \varphi_2$ are any valid LTL formulas. The operator $\mathbf{X}$ (next) is used to define a property $\mathbf{X}\varphi$ that holds if φ holds at the next time step. The binary operator $\mathbf{U}$ (until) is used to specify ordering constraints. The formula, $\varphi_1 \,\mathbf{U}\, \varphi_2$, holds if φ_2 holds at time point in the future, and φ_1 holds until φ_2 first holds. The operators $\neg$ (not), and $\vee$ (or) are identical to propositional logic operators. We also utilize the following abbreviated operators: $\wedge$ (and), $\mathbf{F}$ (eventually), and $\mathbf{G}$ (globally or always). $\mathbf{F}\varphi$ specifies that the formula φ must hold at least once in the future, while $\mathbf{G}\varphi$ specifies that φ must always hold in the future. Consider the Minecraft map depicted in Figure 2. The task of collecting both $wood$ and axe is represented by the LTL formula $\mathbf{F}axe \ \wedge\ \mathbf{F}wood$. The task of collecting $wood$ after collecting axe is represented by the formula $\mathbf{F}(axe \ \wedge\ \mathbf{F}wood)$. Similarly, the task of collecting $wood$ only once axe has been collected is represented by the formula $\mathbf{F}wood \ \wedge\ \neg wood\,\mathbf{U}\,axe$

Every LTL formula can be represented as a Büchi automaton [28, 7] interpreted over an infinite trace of truth values of the propositions used to construct the formula, thus providing an automated translation of a specification into a transition-based representation. We restrict ourselves to the co-safe [14, 18] fragment of LTL that consists of formulas that can be verified by a finite length trace, thus making it ideal for episodic tasks. Camacho et al. [6] showed that each co-safe LTL formula can be translated into an equivalent reward machine [9, 8] $\mathcal{M}_\varphi = \langle \mathcal{Q}_\varphi, q_{0,\varphi}, \mathcal{Q}_{term,\varphi}, \varphi, T_\varphi, R_\varphi\rangle$; where $\mathcal{Q}_\varphi$ is the finite set of states, $q_{0,\varphi}$ is the initial state, $\mathcal{Q}_{term,\varphi}$ is the set of terminal states; $T_\varphi : \mathcal{Q}_\varphi \times 2^{\mathcal{P}} \to \mathcal{Q}_\varphi$ is the deterministic transition function; and $R_\varphi : \mathcal{Q}_\varphi \to \mathbb{R}$ represents the reward accumulated by entering a given state. LTL-Transfer, our proposed algorithm for transferring learned policies to novel LTL specifications, is compatible with all algorithms that generate policies by solving a product MDP of the reward machine $\mathcal{M}_\varphi$ and the task environment.

Options framework: Sutton et al. [23] introduced a framework for incorporating temporally-extended actions, called options, into reinforcement learning. An option $o = \langle \mathcal{I}, \beta, \pi\rangle$ is defined using the initiation set $\mathcal{I}$, which determines the states where the option can be executed; the termination condition β, which determines when option execution ends; and the option policy π. We utilize the options framework to define the task-agnostic skills learned by LTL-Transfer.

3 Related work

Most approaches aimed at extending the reinforcement learning paradigm to temporal tasks rely on the automaton equivalent of the LTL formula to augment the state space and generate an equivalent product MDP. Q-learning for reward machines (Q-RM) [6, 9, 8], geometric-LTL (G-LTL) [17], LPOPL [25] are examples of approaches that extend the environment state-space with the automaton equivalent to the LTL specification. Notably, Jothimurugan et al. [11] proposed DiRL, an algorithm that interleaves graph-based planning on the automaton with hierarchical reinforcement learning to bias exploration towards trajectories that lead to the successful completion of the LTL specification. However, while these approaches exploit the compositional structure of LTL to speed up learning, they do not exploit the compositionality to transfer to novel task specifications. The policy to satisfy a novel LTL formula must be learned from scratch.

A common approach towards generalization in a temporal task setting has been to learn independent policies for each subtask [15, 16, 3, 2] an agent might perform in the environment. When given a new specification, the agent sequentially composes these policies in an admissible order. Consider the Minecraft-inspired grid world depicted in Figure 2 containing $wood$ and axe objects. The subtask-based approaches would train policies to complete subtasks involving reaching each of these objects. In the case of being tasked with the specification $\varphi_{test} = \mathbf{F}wood \wedge (\neg wood\, \mathbf{U}\, axe)$ (i.e. collect $wood$, but do not collect $wood$ until axe is collected), agents trained with the subtask-based approaches would violate the ordering constraint by reaching axe through the grid cells containing $wood$. These approaches rely on additional fine-tuning to correctly satisfy the target task. We propose a general framework for transferring learned policies to novel specifications in a zero-shot setting while preserving the ability to not violate safety constraints.

Our approach draws inspiration from prior works on learning portable skills in Markov domains [12, 10, 4, 5]. These approaches rely on learning a task-agnostic representation of preconditions, constraints, and effects of a skill based on the options framework [23]. We apply this paradigm towards learning portable skills requisite for satisfying temporal specifications.

Kuo et al. [13] proposed learning a modular policy network by composing subnetworks for each proposition and operators. The final policy network is created through the subnetwork modules for a new task specification. Vaezipoor et al. [26] propose learning a latent embedding over LTL formulas using a graph neural network to tackle novel LTL formulas. In contrast, our approach utilizes symbolic methods to identify subpolicies best suited for transfer, thus requiring training on orders of magnitude fewer specifications to achieve comparable results. Finally, Xu and Topcu [29] considered transfer learning between pairs of source and target tasks, while our approach envisions training on a collection of task specifications rather than pairs of source and target tasks.

4 Problem definition

Consider the environment map depicted in Figure 2b. Assume that the agent has trained to complete the specifications to individually collect axe ($\mathbf{F}axe$) and $wood$ ($\mathbf{F}wood$). Now the agent must complete the specification $\mathbf{F}(axe \wedge \mathbf{F}\ wood)$, i.e. first collect axe, then $wood$. Here the agent should identify that sequentially composing the policies for $\mathbf{F}axe$ and $\mathbf{F}wood$ completes the new task (as depicted in blue). Now consider a different test specification $\varphi_2 = \mathbf{F}wood \wedge \neg wood\, \mathbf{U} axe$, i.e. collect $wood$, but avoid visiting $wood$ until axe is collected. Here the agent must realize that policy for $\mathbf{F}axe$ does not guarantee that $wood$ is not visited. Therefore it must not start the task execution using only these learned skills so as to not accidentally violate the ordering constraint. We develop LTL-Transfer to generate such behavior when transferring learned policies to novel LTL tasks. We begin by formally describing the problem setting.

We represent the environment as an MDP without the reward function $\mathcal{M}_{\mathcal{S}} = \langle \mathcal{S}, \mathcal{A}, T_{\mathcal{S}} \rangle$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, and $T_{\mathcal{S}} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ represents the transition dynamics of the environment. We assume that the learning agent does not have access to the transition dynamics. Further, a set $\mathcal{P}$ of Boolean propositions $\boldsymbol{\alpha}$ represents the facts about the environment, and a labeling function $L : \mathcal{S} \to 2^{\mathcal{P}}$ maps the state to these Boolean propositions. These Boolean

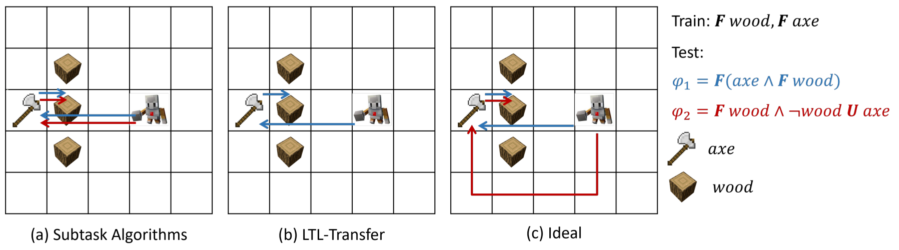


Figure 2: An example 5×5 map in a Minecraft-like grid world. The agent is assumed to have trained on the two training specifications, and is expected to satisfy φ_1 and φ_2. Figure 1a depicts the trajectories adopted by an agent using a subtask-based algorithm (blue for φ_1, red for φ_2). Figure 1b depicts the trajectories followed by LTL-Transfer, our proposed algorithm. Note that LTL-Transfer does not start the task execution for φ_2, as the training task policies do not guarantee the preservation of the ordering constraint. Figure 1c depicts the optimal trajectories for φ_1 and φ_2.

propositions are the compositional building blocks for defining the tasks that can be performed within the environment $\mathcal{M}_{\mathcal{S}}$.

We assume that a task within the environment $\mathcal{M}_{\mathcal{S}}$ is defined by a linear temporal logic (LTL) formula φ, and that the agent is trained on a set of training tasks $\Phi_{train} = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$. We further assume that these policies were learned using a class of reinforcement learning algorithms that operate on a product MDP composed of the environment $\mathcal{M}_{\mathcal{S}}$, and the automaton representing the non-Markov LTL task specification. Q-RM [6, 9, 8], G-LTL [17], LPOPL [25] are examples of such algorithms. LPOPL explicitly allows for sharing policies for specifications that share progression states; therefore, as the baseline best suited for transfer in a zero-shot setting, we choose LPOPL as our learning algorithm of choice.

LTL-Transfer operates in two stages. In the first stage, it accepts the set of training tasks Φ_{train} and the learned policies, and outputs the set of task-agnostic, portable options $\mathcal{O}_e$. In the second stage, given a novel task specification φ_{test} and the set of options $\mathcal{O}_e$, LTL-Transfer identifies and executes a sequence of options to satisfy φ_{test}.

5 LTL-Transfer with transition-centric options

An LTL specification $\varphi \in \Phi_{train}$ to be satisfied is represented as the reward machine $\mathcal{M}_\varphi = \langle \mathcal{Q}_\varphi, q_{0,\varphi}, \mathcal{Q}_{term,\varphi}, \varphi, T_\varphi, R_\varphi \rangle$. This specification must be satisfied by the agent operating in an environment $\mathcal{M}_{\mathcal{S}} = \langle \mathcal{S}, \mathcal{A}, T_{\mathcal{S}} \rangle$. The policy learned by LPOPL is Markov with respect to the environment states $\mathcal{S}$ for a given RM state, i.e. the subpolicy to be executed in state $q \in \mathcal{Q}_\varphi$, $\pi_q^\varphi : \mathcal{S} \to \mathcal{A}$.

An option o_q^φ is executed in the reward machine (RM) state q. Our insight is that each of these options triggers a transition in the reward machine on a path that leads towards an acceptance state, and these transitions may occur in multiple tasks. There might be multiple paths through the reward machine to an accepting state; therefore, the target transition of an option o_q^φ is conditioned on the environment state where the option execution was initiated. We propose recompiling each state-centric option into multiple transition-centric options by partitioning the initiation set of the state-centric option based on the transition resulting from the execution of the option policy from the starting state. Each resulting transition-centric option will maintain the truth assignments of $\mathcal{P}$ to ensure self-transitions till it achieves the truth assignments required to trigger the intended RM transition. These transition-centric options are portable across different formulas. We describe our proposed algorithm in Section 5.1.

Given a novel task specification $\varphi_{test} \notin \Phi_{train}$, the agent first constructs a reward machine representation of the specification, $\mathcal{M}_{\varphi_{test}}$, then identifies a path through the reward machine that can be traversed by a sequential composition of the options from the set of transition-centric options $\mathcal{O}_e$. A key feature of our transfer algorithm is that it is sound and terminating, i.e. if it returns a solution with success, that task execution will satisfy the task specification. Further, it is guaranteed to terminate in finite time if it does not find a sequence of options that can satisfy a given task. We describe the details of this planning algorithm in Section 5.2.

The key advantage of this approach is that the compilation of options can be computed offline for any given environment, and the options can then be transferred to novel specifications. Thus learning to satisfy a limited number of LTL specifications can help satisfy a wide gamut of unseen LTL specifications.

5.1 Compilation of transition-centric options

The policy learned by LPOPL to satisfy a specification φ identifies the current reward machine state $q \in \mathcal{Q}_\varphi$ the task is in and executes a Markov policy π_q^φ till the state of the reward machine progresses. This subpolicy can be represented as an option, $o_q^\varphi = \langle \mathcal{S}, \beta_{e_{q,q}^\varphi}, \pi_q^\varphi \rangle$; where the initiation set is the entire state-space of the task environment; the option terminates when the truth assignments of the propositions $\boldsymbol{\alpha}$ do not satisfy the self-transition, represented by the Boolean function β_e defined as follows,

$$\beta_e = \begin{cases} 1, & \text{if } L(s) \not\models e \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

A transition-centric option, $o_{e_1 . e_2}$, executes a Markov policy such that it ensures that the truth assignments of $\mathcal{P}$ satisfy the self transition formula e_1 at all time steps until the policy yields a truth assignment that satisfies e_2. A transition-centric option is defined by the following tuple:

$$o_{e_1, e_2} = \langle \mathcal{S}, \beta_{e_1}, \pi, e_1, e_2, f_{e_2} \rangle. \tag{3}$$

Here, the initiation set represents the entire environment state-space S; the termination condition is defined by the dissatisfaction of the termination condition as represented by β_{e_1}; the option executes the Markov policy $\pi : \mathcal{S} \to \mathcal{A}$; e_1 and e_2 represent the self-transition and the target edge formulas respectively; and $f_{e_2} : \mathcal{S} \to [0, 1]$ represents the probability of completing the target edge e_2 when starting from $s \in \mathcal{S}$.

Algorithm 1 describes our approach to compiling each state-centric option o_q^φ into a set of transition-centric options. If $\mathcal{E}$ is the set of pairs of self and outgoing edge formulas from state q of the reward machine, then executing the option's policy π_q^φ results in a distribution over the outgoing edges $\{e_{q,q'}^\varphi : q' \text{ is out-neighbor of } q\}$ conditioned on the environment state $s \in \mathcal{S}$ where the option execution was initiated.

Thus the distribution $f_{e_{q,q'}^\varphi}$ acts as a soft segmenter of the state-space $\mathcal{S}$. $f_{e_{q,q'}^\varphi}$ is estimated by sampling rollouts from all possible environment states in discrete domains, or can be learnt using sampling-based methods [4, 5] in continuous domains. Each state-centric option o_q can be compiled into a set of transition-options, $\left\{ o_{e_{q,q}^\varphi, e_{q,q'}^\varphi} : q \in \mathcal{Q}_\varphi, q' \text{is out-neighbor of } q \right\}$.

Algorithm 1 Compile state-centric options to transition-centric options

```
1: function COMPILE($\mathcal{M}_\mathcal{S}, \Phi_{train}, \mathcal{O}_q$)    ▷ Environment, training specifications, and the learned state-centric options
2:     $\mathcal{O}_e \leftarrow \emptyset$
3:     for $\varphi \in \Phi_{train}$ do
4:         $\mathcal{M}_\varphi \leftarrow$ GENERATERM($\varphi$)
5:         $\mathcal{O}_q^\varphi \leftarrow \left\{ o_q^{\varphi'} : \varphi' = \varphi, o_q^{\varphi'} \in \mathcal{O}_q \right\}$    ▷ All state-centric options associated with task $\varphi$
6:         for $o_q^\varphi = \langle \mathcal{S}, \beta_{e_1}^\varphi, \pi_q^\varphi \rangle \in \mathcal{O}_q^\varphi$ do
7:             $\mathcal{E} \leftarrow \left\{ (e_{q,q}^\varphi, e_{q,q'}^\varphi) : e_{q,q}^\varphi \text{ is the self edge, } q' \text{ is an out-neighbor of } q \right\}$
8:             for $s \in \mathcal{S}$ do
9:                 Generate $N_r$ rollouts from $s$ with $\pi_q^\varphi$
10:                Record edge transition frequencies $n_s(e_2) \ \forall \ (e_1, e_2) \in \mathcal{E}$
11:                $f_{e_{q,q'}^\varphi}(s) \leftarrow \frac{n_s(e_{q,q'}^\varphi)}{N_r} \ \forall \ q' \in \{q' : q' \text{ is an out-neighbor of } q\}$
12:            $\mathcal{O}_e^{q,\varphi} \leftarrow \left\{ o_e^{q,\varphi} = \langle \mathcal{S}, \beta_{e_{q,q}^\varphi}, \pi_q^\varphi, e_{q,q}^\varphi, e_{q,q'}^\varphi, fe_{q,q'}^\varphi \rangle \right\}$    ▷ All transition-centric options from state-centric option $o_q^\varphi$
13:            $\mathcal{O}_e \leftarrow \mathcal{O}_e \cup \mathcal{O}_e^{q,\varphi}$
14:    return $\mathcal{O}_e$
```

5.2 Transferring to novel task specifications

Our proposed algorithm for composing the transition-centric options in the set $\mathcal{O}_e$ to solve a novel task specification φ_{test} is described in Algorithm 2. Once the reward machine for the test task specification is generated, Line 3 examines each edge of the reward machine, and identifies the transition-centric options that can achieve the edge transition while maintaining the self transition $e^{\varphi_{test}}_{q',q'}$, where q' is the source node of the edge; if no such option is identified, we remove this edge from the reward machine. Line 7 identifies all paths in the reward machine from the current node to the accepting node of the RM. Lines 8 and 9 construct a set of all available options that can potentially achieve an outgoing transition from the current node to a node on one of the feasible paths to the goal state.

The agent then executes the option with the highest probability of achieving the intended edge transition determined by function f (Lines 12 and 13). Note that the termination condition for the option, $o^*_{e_1,e_2}$ is satisfied when either the option's self transition condition is violated, i.e. $L(s) \not\models e_1$, or when it progresses to a new state of the reward machine $\mathcal{M}_\varphi$.

If the option fails to progress the reward machine, it is deleted from the set (Line 15), and the next option is executed. If at any point, the set of executable options is empty without having reached the accepting state $q^\top$, Algorithm 2 exits with a failure (Line 17). If the reward machine progresses to $q^\top$, it exits with success.

Algorithm 2 Zero-shot transfer to test task φ^*

```
1: function TRANSFER($\mathcal{M}_\mathcal{S}, \varphi^*, \mathcal{O}_e$)
2:     $\mathcal{M}^*_\varphi \leftarrow$ GENERATE_RM($\varphi^*$)
3:     $\mathcal{M}^*_\varphi \leftarrow$ PRUNE($\mathcal{M}_{\varphi^*}$)
4:     $s \leftarrow$ INITIALIZE($\mathcal{M}_\mathcal{S}$)
5:     $q \leftarrow q_{0,\varphi^*}$
6:     while $q \neq q^\top$ do        ▷ $q^\top \in \mathcal{Q}_{term,\varphi^*}$ is the accepting state for the underlying task specification.
7:         $P \leftarrow \left\{p_i \ : \ p_i = [e_0, \dots e_{n^i}]\right.$ are paths connecting $q$ and $q^\top$ in $\left.\mathcal{M}_{\varphi^*}\right\}$
8:         $\forall p \in P : \mathcal{O}_{p[0]} = \left\{o_{e_1,e_2} : \text{MATCHEDGE}((e_1, e_2), (e^{\varphi^*}_{q,q}, p[0])), o_{e_1,e_2} \in \mathcal{O}_e\right\}$        ▷ Edge options for the first edge in each path
9:         $\mathcal{O}_{[0]} = \bigcup_p \mathcal{O}_{p[0]}$
10:        $\langle s', q' \rangle \leftarrow \langle s, q \rangle$
11:        while $\mathcal{O}_{[0]} \neq \emptyset$ and $q' = q$ do
12:            $o^* \leftarrow \arg\max_{o_{e_1,e_2} \in \mathcal{O}_{[0]}} f_{e2}(s)$        ▷ Select option most likely to complete the transition
13:            $\langle s', q' \rangle \leftarrow$ EXECUTE($\pi^*$)        ▷ $\pi^*$ is the policy corresponding to the option
14:            if $q' = q$ then
15:                $\mathcal{O}_{[0]} \leftarrow \mathcal{O}_{[0]} \setminus o^*$        ▷ If $o^*_e$ does not induce progression, delete it
16:        if $q' = q$ then
17:            return Failure
18:        else
19:            $\langle s, q \rangle \leftarrow \langle s', q' \rangle$
20:    return Success
```

5.3 Matching transition-centric options to reward machine edges

The edge matching conditions identify whether a given transition-centric option can be applied safely to transition along an edge of the reward machine on a feasible path. Here we propose two edge matching conditions, *constrained* and *relaxed*, that both ensure that the task execution does not fail due to an unsafe transition. The edge matching conditions are used to prune the reward machine graph to contain only the edges with feasible options available (Line 3) and enumerate feasible options from a given reward machine state (Line 8). We use a propositional model counting approach [27] to evaluate the edge matching conditions. We propose the following two edge matching conditions (Further details of the implementation of the edge matching conditions are provided in the supplementary material):

Constrained Given a test specification φ_{test}, where the task is in the state q, the self-transition edge is $e^{\varphi_{test}}_{q,q}$ and the targeted edge transition is $e^{\varphi_{test}}_{q,q'}$, we must determine if the transition-centric option o_{e_1,e_2} matches the required transitions. The *Constrained* edge matching criterion ensures that every truth assignment that satisfies the outgoing edge of the option, e_2, also satisfies the targeted transition for the test specification $e^{\varphi_{test}}_{q,q'}$. Similarly, every truth assignment that satisfies the self-transition edge of the option e_1 must also satisfy the self-transition formula $e_{q,q}^{\varphi_{test}}$. This strict requirement reduces the applicability of the learned options for satisfying novel specifications but ensures that the targeted edge is always achieved.

Relaxed For the *Relaxed* edge matching criterion, the self edges e_1 and $e_{q,q}^{\varphi_{test}}$ must share satisfying truth assignments, so must the targeted edges e_2 and $e_{q,q'}^{\varphi_{test}}$. However, it allows the option to have valid truth assignments that may not satisfy the intended outward transition; yet none of those truth assignments should trigger a transition to an unrecoverable failure state $q^{\perp}$ of the reward machine. Further, all truth assignments that terminate the option must not satisfy the self-transition condition for the test specification. The *Relaxed* edge matching conditions can retrieve a greater number of eligible options.

6 Experiments

We evaluated the LTL-Transfer algorithm in the Minecraft-inspired domains[2] commonly seen in research into compositional reinforcement learning and integration of temporal logics with reinforcement learning [2, 25, 11, 3]. In these domains, the task specifications comprise a set of subtasks that the agent must complete and a list of precedence constraints defining the admissible orders in which the subtasks must be executed. These specifications belong to the class of formulas that form the support of the prior distributions proposed by Shah et al. [22].

Our experiments are aimed at evaluating the following hypotheses:

1. **H1:** Both the *Constrained* and *Relaxed* edge matching conditions should exceed LPOPL's capability to transfer to novel specifications. Note that while LPOPL was not explicitly developed to transfer to novel specifications in a zero-shot setting, it can satisfy specifications that are a progression of one of the formulas that the agent was trained on.
2. **H2:** *Relaxed* edge matching criterion will result in a greater success rate than the *Constrained* criterion.
3. **H3:** It is easier to transfer learned policies for LTL formulas conforming to certain templates (Section 6.2).
4. **H4:** Training with formulas conforming to certain formula templates leads to a greater success rate when transferring to all specification types.

6.1 Task Environment

We implement LTL-transfer[3] within a Minecraft-inspired discrete grid-world domain [2, 25]. Each grid cell can be occupied by one of nine object types, or it may be vacant; note that multiple instances of an object type may occur throughout the map. The agent can choose to move along any of the four cardinal directions, and the outcome of these actions is deterministic. An invalid action would result in the agent not moving at all. A given task within this environment involves visiting a specified set of object types in an admissible order determined by ordering constraints. The different types of ordering constraints are described in Section 6.2. Task environment maps are similar to the 5×5 maps depicted in Figure 2; however, all the maps used for evaluation were 19×19.

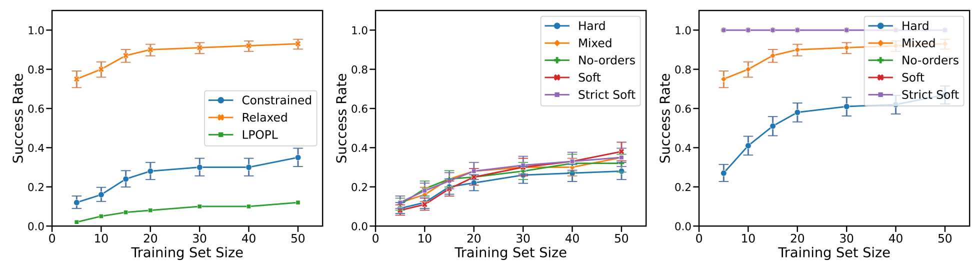
6.2 Specification Types

We considered the following three types of ordering constraints for a comprehensive evaluation of transferring learned policies across different LTL specifications. Each constraint is defined on a binary pair of propositions a and b, and without loss of generality, we assume that a should precede b. The three types of constraints are as follows:

1. **Hard:** Hard orders occur when b must never be true before a. In LTL, this property can be expressed through the formula $\neg b \ \mathbf{U} \ a$.
2. **Soft:** Soft orders allow b to occur before a as long as b happens at least once after a holds for the first time. Soft orders are expressed in LTL through the formula $\mathbf{F}(a \wedge \mathbf{F}b)$.

[2]We used the version by Toro Icarte et al. [25] `https://bitbucket.org/RToroIcarte/lpopl`

[3]Code: `https://github.com/jasonxyliu/ltl_transfer`



(a) (b) (c)

Figure 3: Figure 3a depicts the success rate on the *mixed* test set after training on the *mixed* training set of various sizes for the LPOPL baseline and for the two edge-matching criteria. Figure 3b depicts the success rate of the agent trained on *mixed* training sets of various sizes using LTL-Transfer with the *Constrained* edge-matching criterion when transferring to test sets of various specifications types. Figure 3c depicts the success rates with the *Relaxed* edge-matching criterion. Note that the error bars depict the 95% credible interval if the successful transfer was modeled as a Bernoulli distribution.

3. **Strictly Soft:** Strictly soft ordering constraints are similar to soft orders; however, b must be true strictly after a first holds. Thus a and b holding simultaneously would not satisfy a strictly soft order. Strictly soft orders are expressed in LTL through the formula $\mathbf{F}(a \wedge \mathbf{XF}b)$

We sampled five training sets: *hard*, *soft*, *strictly soft*, *no-orders*, and *mixed*; with 50 formulas each that represent different specification types. The sub-tasks to be completed and the ordering constraints were sampled from the priors proposed by Shah et al. [22]. All ordering constraints within the *hard*, *soft*, and *strictly soft* training sets were expressed through the respective templates described here. There were no ordering constraints to be satisfied for the *no-orders*. In the *mixed* training set, each binary precedence constraint was expressed as one of the three ordering types described in 6.2.

In addition to the training set, we sampled a test set of 100 formulas for each set type. This mimics the real-world scenario where the agent would train on a few specifications but might be expected to satisfy a wide array of specifications during deployment.

6.3 Experiment configurations

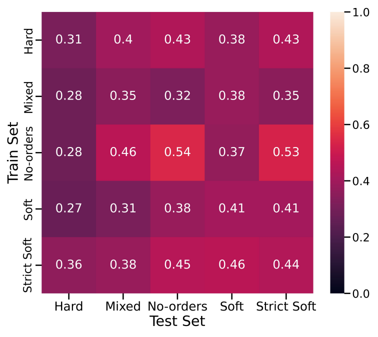
For each experimental run, we specified the training set type and size and the test set type. All experiments were conducted on four different grid world maps. The evaluation metrics include the success rate on each of the test set specifications. We logged the reason for any failed run.

The precomputations for compiling the set of edge-centric options were computed on a high-performance computing (HPC) cluster hosted by our university. As the compilation of state-centric options into transition-centric options allows for large-scale parallelism with no interdependency, we were able to share the workload among a large number of CPU cores.

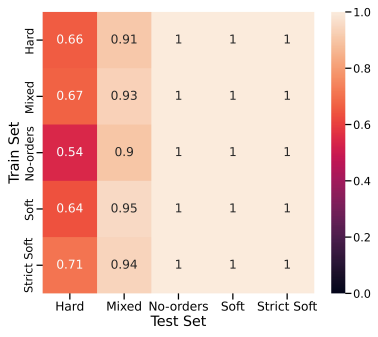
7 Results and Discussion

Comparison with LPOPL LPOPL's use of progressions and the multi-task learning framework allow it to handle tasks that lie within the progression set of the training formulas. To compare the performance of LTL-Transfer and LPOPL, we trained each of them on *mixed* training sets of varying sizes and evaluated their success rate on *mixed* test set. Figure 3a depicts the success rate of LTL-Transfer (orange and blue lines) and LPOPL (green line). The error bars represent the 95% credible interval if the success rate was modeled as the parameter of a Bernoulli distribution with a conjugate beta prior. Note that LTL-Transfer exceeds the performance of LPOPL in zero-shot transfer to novel specifications using both the *Constrained* and *Relaxed* edge matching criteria (Section 5.3) thus supporting **H1**. By training on 50 specifications of the *mixed* type, LTL-Transfer with the *Relaxed* edge matching criterion can complete more than 90% of unseen task specifications.

Effect of edge matching criterion Next, we trained our agent on mixed specification types of varying sizes and used LTL-Transfer to transfer the learned policies to complete the specifications



(a)



(b)

Figure 4: Figure 4a depicts the heatmap of success rates with various training and test specification types with the *Constrained* edge matching criterion. Similarly, Figure 4b depicts the heatmap with the *Relaxed* edge matching criterion.

in all five test sets. The success rates with the *Constrained* edge matching criterion are depicted in Figure 3b, while those for the *Relaxed* edge matching criterion are depicted in Figure 3c. We note that the *Relaxed* edge matching criterion is capable of successfully transferring to a larger number of novel specifications across all specification types, thus supporting **H2**

Relative difficulty of specification type Figure 3b indicates that the different specifications are equally difficult to transfer learned policies to when using the *Constrained* edge matching criterion. However, Figure 3c indicates that with the *Relaxed* edge matching criterion, LTL-Transfer is capable of transferring to novel specifications with Soft or Strictly Soft orders after training on very few specifications. It also indicates that specifications with Hard orders are the most difficult to transfer to. Therefore **H3** is supported only for the *Relaxed* edge matching criterion, and not for the *Constrained* criterion.

Transferring across specification type Finally, we evaluate whether certain specification types are more capable of transferring to all specification types by training our agent on different specification types and attempting to transfer those policies to other specification types. Figure 4a depicts the heatmap of success rates obtained by training the agent on 50 specifications of the type indicated by the row and transferring it to the test set of specification types indicated by the column while using the *Constrained* edge matching criterion. Similarly, Figure 4b depicts the success rates using the *Relaxed* edge matching criterion. Note that no single specification type proved to be the best training set, thus providing evidence against **H4**.

Note that in all of our runs, the agent never violated a constraint leading to an unrecoverable failure, which is crucial in safety-critical applications. The causes for failure to transfer included cases where there was no feasible path to an accepting state with the set of options, or the agent attempted all available options without progressing the task.

8 Robot Demonstrations

We demonstrate LTL-Transfer on Spot [1], a quadruped mobile manipulator, in a household environment where the robot can fetch and deliver objects while navigating through the space. The robot was trained on 2 LTL tasks $\neg desk_a$ $\mathbf{U}$ $book \wedge \mathbf{F}desk_a$ and $\neg desk_b$ $\mathbf{U}$ $juice \wedge \mathbf{F}desk_b$, then transferred the learned skills to 8 combinations of soft-ordering tasks $\mathbf{F}(obj_* \wedge \mathbf{F}(desk_* \wedge \mathbf{F}(obj_* \wedge \mathbf{F}desk_*)))$ in zero-shot fashion. For the tasks that LTL-Transfer cannot transfer (e.g. $\neg desk_a$ $\mathbf{U}$ $book$ $\wedge$ $\neg juice$ $\mathbf{U}$ $desk_a$ $\wedge$ $\neg desk_b$ $\mathbf{U}juice$ $\wedge$ $\mathbf{F}desk_b$), the robot, as expected, does not start execution thus avoids violations of any constraints [4].

Please see the supplementary material for more details about this environment, the training and test tasks.

[4]Video: `https://youtu.be/FrY7CWgNMBk`

9 Conclusion

We introduced LTL-Transfer, a novel algorithm that leverages the compositionality of linear temporal logic to solve a wide variety of novel, unseen LTL specifications. It segments policies from training tasks into portable, task-agnostic transition-centric options that can be reused for any task. We demonstrate that LTL-Transfer can solve over 90% of unseen task specifications in our Minecraft-inspired domains while training on only 50 specifications. We further demonstrated that LTL-Transfer never violated any safety constraints and aborted task execution when no feasible solution was found.

LTL-Transfer enables the possibility of maximally transferring the learned policies of the robot to new tasks. We envision further developing LTL-Transfer to incorporate long-term planning and intra-option policy updates to generate not just satisfying but optimal solutions to novel tasks.

Acknowledgments

This work was supported by NSF under grant awards DGE-2040433 and CNS-2038897, and ONR under grant awards N00014-17-1-2699 and N00014-21-1-2584.

References

[1] Spot® - the agile mobile robot. URL `https://www.bostondynamics.com/products/spot`.

[2] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *International Conference on Machine Learning*, pages 166–175. PMLR, 2017.

[3] Brandon Araki, Xiao Li, Kiran Vodrahalli, Jonathan Decastro, Micah Fry, and Daniela Rus. The logical options framework. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 307–317. PMLR, 18–24 Jul 2021. URL `https://proceedings.mlr.press/v139/araki21a.html`.

[4] Akhil Bagaria and George Konidaris. Option discovery using deep skill chaining. In *International Conference on Learning Representations*, 2019.

[5] Akhil Bagaria, Jason Senthil, Matthew Slivinski, and George Konidaris. Robustly learning composable options in deep reinforcement learning. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence*, 2021.

[6] Alberto Camacho, Rodrigo Toro Icarte, Toryn Q Klassen, Richard Anthony Valenzano, and Sheila A McIlraith. Ltl and beyond: Formal languages for reward function specification in reinforcement learning. In *IJCAI*, volume 19, pages 6065–6073, 2019.

[7] Rob Gerth, Doron Peled, Moshe Y Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *International Conference on Protocol Specification, Testing and Verification*, pages 3–18. Springer, 1995.

[8] Rodrigo Toro Icarte, Toryn Klassen, Richard Valenzano, and Sheila McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. In *International Conference on Machine Learning*, pages 2107–2116. PMLR, 2018.

[9] Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano, and Sheila A McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research*, 73:173–208, 2022.

[10] Steven James, Benjamin Rosman, and George Konidaris. Learning portable representations for high-level planning. In *International Conference on Machine Learning*, pages 4682–4691. PMLR, 2020.

[11] Kishor Jothimurugan, Suguman Bansal, Osbert Bastani, and Rajeev Alur. Compositional reinforcement learning from logical specifications. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

[12] George Dimitri Konidaris and Andrew G Barto. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, pages 895–900, 2007.

[13] Yen-Ling Kuo, Boris Katz, and Andrei Barbu. Encoding formulas as deep networks: Reinforcement learning for zero-shot execution of ltl formulas. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5604–5610. IEEE, 2020.

[14] Orna Kupferman and Moshe Y Vardi. Model checking of safety properties. *Formal methods in system design*, 19(3):291–314, 2001.

[15] Borja G León, Murray Shanahan, and Francesco Belardinelli. Systematic generalisation through task temporal logic and deep reinforcement learning. *arXiv preprint arXiv:2006.08767*, 2020.

[16] Borja G León, Murray Shanahan, and Francesco Belardinelli. In a nutshell, the human asked for this: Latent goals for following temporal specifications. *arXiv preprint arXiv:2110.09461*, 2021.

[17] Michael L Littman, Ufuk Topcu, Jie Fu, Charles Isbell, Min Wen, and James MacGlashan. Environment-independent task specifications via gltl. *arXiv preprint arXiv:1704.04341*, 2017.

[18] Zohar Manna and Amir Pneuli. A hierarchy of temporal properties. In *ACM symposium on Principles of distributed computing*, 1990.

[19] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL `https://doi.org/10.7717/peerj-cs.103`.

[20] Roma Patel, Ellie Pavlick, and Stefanie Tellex. Grounding language to non-markovian tasks with no supervision of task specifications. In *Robotics: Science and Systems*, 2020.

[21] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. ieee, 1977.

[22] Ankit Shah, Pritish Kamath, Julie A Shah, and Shen Li. Bayesian inference of temporal task specifications from demonstrations. *Advances in Neural Information Processing Systems*, 31, 2018.

[23] Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1): 181–211, 1999. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(99)00052-1. URL `https://www.sciencedirect.com/science/article/pii/S0004370299000521`.

[24] M.E. Taylor, , and P. Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009.

[25] Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Valenzano, and Sheila A. McIlraith. Teaching multiple tasks to an rl agent using ltl. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 2018. to appear.

[26] Pashootan Vaezipoor, Andrew C Li, Rodrigo A Toro Icarte, and Sheila A Mcilraith. LTL2action: Generalizing LTL instructions for multi-task RL. In *International Conference on Machine Learning*, pages 10497–10508. PMLR, 2021.

[27] Leslie G Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8 (2):189–201, 1979.

[28] Moshe Y Vardi. An automata-theoretic approach to linear temporal logic. *Logics for concurrency*, pages 238–266, 1996.

[29] Zhe Xu and Ufuk Topcu. Transfer of temporal logic formulas in reinforcement learning. In *IJCAI: proceedings of the conference*, volume 28, page 4010. NIH Public Access, 2019.

10 Appendix

10.1 Edge matching criteria

LTL-Transfer uses a set of edge matching conditions to prune the infeasible edges from the reward machine (RM) graph given a set of transition-centric options and to determine the candidate options given the current RM state and a particular outgoing edge to target. Below, we present two edge matching conditions, *constrained* and *relaxed*, that both ensure progression of the reward machine towards and the transition leading to non-failure states.

Without the loss of generality, we assume the test specification is φ_{test}, the test task is in RM state q, where the self-transition edge is $e^{\varphi_{test}}_{q,q}$ and the targeted edge transition is $e^{\varphi_{test}}_{q,q'}$. We are determining the applicability of the transition-centric option o_{e_1,e_2}.

10.1.1 Constrained edge matching criterion

The *constrained* edge matching criterion ensures every truth assignment that satisfies the self transition edge $e^{\varphi_{test}}_{q,q}$ of RM also satisfies the self transition edge e_1 of the transition-centric option, and the same condition holds for the outgoing edge $e^{\varphi_{test}}_{q,q'}$ of RM and the outgoing edge e_2 of the option. Mathematically, the following equation must evaluate to true to declare a constrained match, where asg and asg' represent truth assignments, $sat(f, asg)$ is a function that evaluates the Boolean formula f on the truth assignment asg, and $sat\_models(f)$ returns all truth assignments that satisfy the Boolean formula f. We used the Sympy library for logic operations [19].

$$sat(e_1, asg)\ \wedge sat(e_2, asg')\ \forall asg \in sat\_models(e^{\varphi_{test}}_{q,q}),\ \forall asg' \in sat\_models(e^{\varphi_{test}}_{q,q'}) \tag{4}$$

10.1.2 Relaxed edge matching criterion

The *relaxed* edge matching criterion ensures that the self transition edges e_1 and $e^{\varphi_{test}}_{q,q}$ share satisfying truth assignments, so are the outgoing edges e_2 and $e^{\varphi_{test}}_{q,q'}$. We let the failure edge $e^{\perp}$ be the edge from the current RM state q to the failure state $q^{\perp}$ if one exists. To prevent the selection of a transition-centric option o_{e_1,e_2} that violates constraints, we enforce that both self and outgoing edges of the option, e_1 and e_2, must not share any satisfying truth assignment with the failure edge $e^{\perp}$ if one exists. Lastly, to guarantee RM progression after applying the transition-centric option, the *relaxed* edge matching condition ensures that the outgoing edge e_2 of the option must share no satisfying truth assignments with the self transition edge $e^{\varphi_{test}}_{q,q}$ of the RM. Mathematically, the following equation must evaluate to true to declare a relaxed match, where $sat(f)$ is a function that determines if any truth assignment exists to satisfy the Boolean formula f.

$$sat(e_1 \wedge e^{\varphi_{test}}_{q,q}) \wedge sat(e_2 \wedge e^{\varphi_{test}}_{q,q'}) \wedge \neg sat(e_1 \wedge e^{\perp}) \wedge \neg sat(e_2 \wedge e^{\perp}) \wedge \neg sat(e_2 \wedge e^{\varphi_{test}}_{q,q}) \tag{5}$$

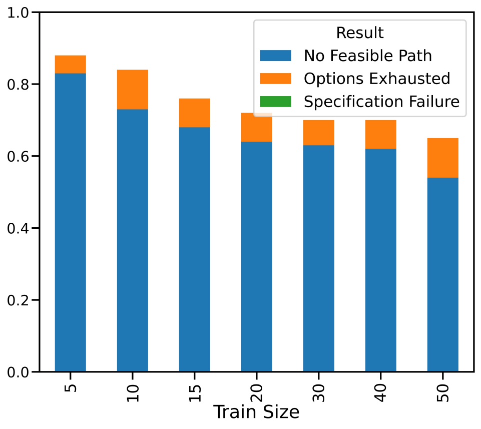
10.2 Full Results

Cause of failure: As described in our draft, we logged the reason for failure for each unsuccessful transfer attempt as one of three possible causes: *specification failure*, where the agent violates a constraint and the reward machine is progressed to an unrecoverable state; *no feasible path*, where there are no matched transition-centric options for paths connecting the start state to an accepting state; *options exhausted*, where there are no further transition-centric options available to the agent to further progress the state of the task.

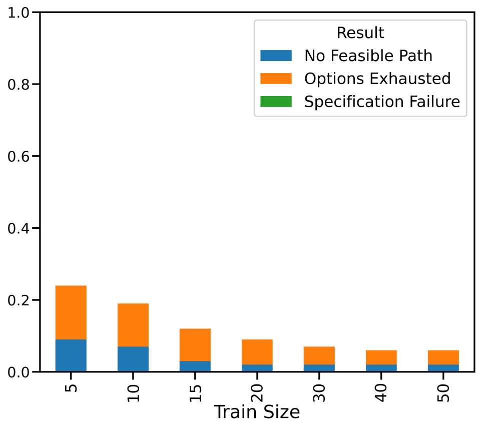
Figure 5 depicts the relative frequency of the failure modes when the agent is trained and tested on *mixed* task specifications. Note that with the *Constrained* edge-matching criterion, absence of feasible paths connecting the start and the accepting state is the primary reason for failure (Figure 5a, whereas with the *Relaxed* edge-matching criterion, the agent utilizing all available safe options without progressing the task is the primary reason for failure (Figure 5b).

Learning curve for various training datasets: Next, we present the results for learning curve of the success rate when transferring policies learned on different specification types.

The learning curves for training on formulas from the *Hard* training set with both the edge matching criteria are depicted in Figure 6.



(a) Constrained



(b) Relaxed

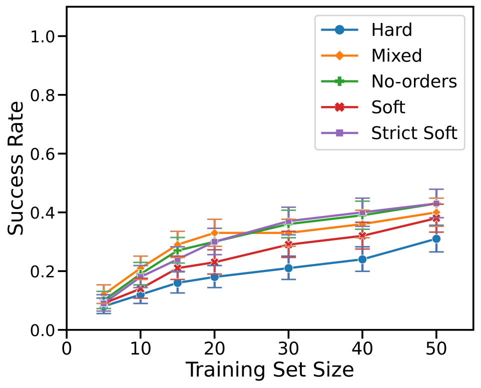
Figure 5: Reasons for failed task executions for agents trained and evaluated on *Mixed* task specification datasets. Note that all values are depicted in fractions.

The learning curves for training on formulas from the *Soft* training set with both the edge matching criteria are depicted in Figure 7.

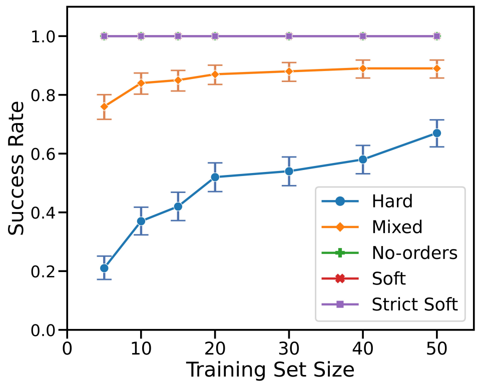
The learning curves for training on formulas from the *Strictly Soft* training set with both the edge matching criteria are depicted in Figure 8.

The learning curves for training on formulas from the *No Orders* training set are being generated at the time of submission, and are expected to share nearly identical trends as the learning curves from the other training sets given the completed data points. We will include the plots in the final version of the paper.

Note that for training on each of the specification types, the learning curve trends are nearly identical to the learning curves on training with *Mixed* specification types as depicted in Figure 3 in the main draft. *Hard* specification types remain the most challenging specification ordering types to transfer to.



(a) Constrained

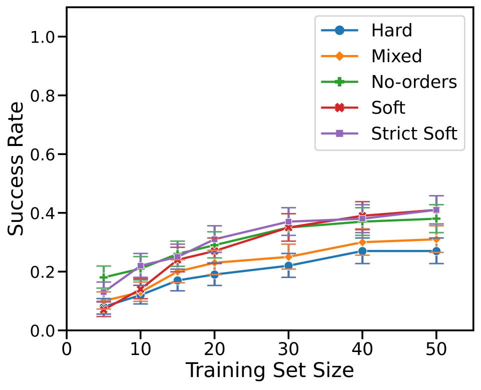


(b) Relaxed

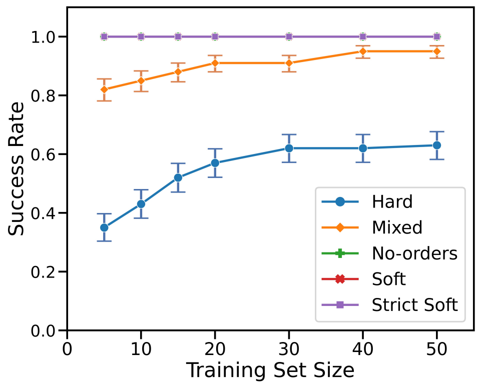
Figure 6: Figure 6a depicts the success rate of the agent trained on *Hard* training sets of various sizes using LTL-Transfer with the *constrained* edge-matching criterion when transferring to test sets of various specifications types. Figure 6b depicts the success rates with the *relaxed* edge-matching criterion. Note that the error bars depict the 95% credible interval if the successful transfer was modeled as a Bernoulli distribution.

10.3 Selected Solution Trajectories

Consider the case with *mixed* training set with 5 formulas on *map 0*. The training formulas are:

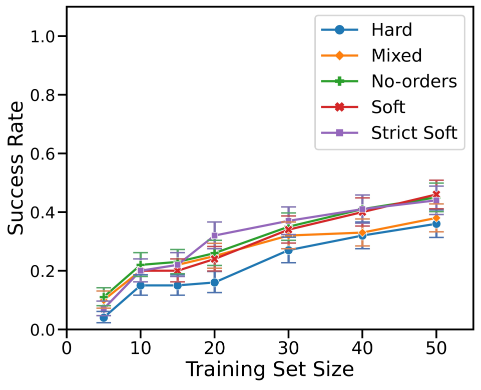


(a) Constrained

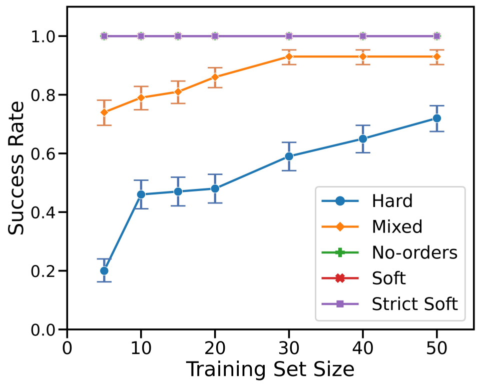


(b) Relaxed

Figure 7: Figure 7a depicts the success rate of the agent trained on *Soft* training sets of various sizes using LTL-Transfer with the *constrained* edge-matching criterion when transferring to test sets of various specifications types. Figure 7b depicts the success rates with the *relaxed* edge-matching criterion. Note that the error bars depict the 95% credible interval if the successful transfer was modeled as a Bernoulli distribution.



(a) Constrained



(b) Relaxed

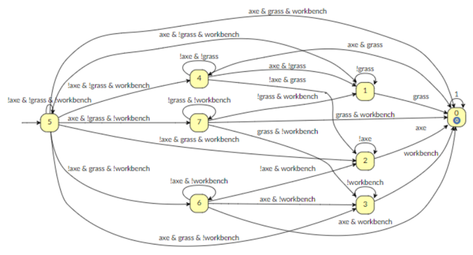
Figure 8: Figure 8a depicts the success rate of the agent trained on *Strictly Soft* training sets of various sizes using LTL-Transfer with the *constrained* edge-matching criterion when transferring to test sets of various specifications types. Figure 8b depicts the success rates with the *relaxed* edge-matching criterion. Note that the error bars depict the 95% credible interval if the successful transfer was modeled as a Bernoulli distribution.

- $\mathbf{F} grass \wedge \mathbf{F} shelter \wedge \mathbf{F}(wood \wedge \mathbf{XF} workbench)$
- $\mathbf{F} toolshed \wedge \mathbf{F} workbench \wedge \mathbf{F} shelter \wedge (\neg toolshed\, \mathbf{U}\, shelter) \wedge \mathbf{F}(grass \wedge \mathbf{F} bridge)$
- $\mathbf{F} toolshed \wedge \mathbf{F}(shelter \wedge \mathbf{F}(axe \wedge \mathbf{F} wood))$
- $\mathbf{F} iron \wedge \mathbf{F}(shelter \wedge \mathbf{XF}(bridge \wedge \mathbf{XF} factory))$
- $\mathbf{F} factory$

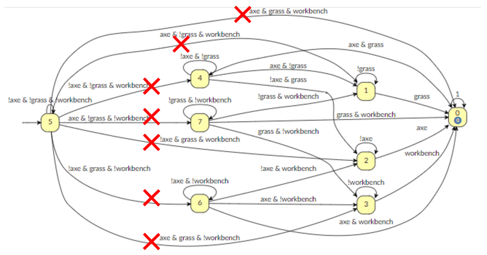
One of the *Mixed* test formulas was $\varphi_{test} = \mathbf{F} workbench \wedge \mathbf{F} grass \wedge \mathbf{F} axe$. The reward machine for this task specification is depicted in Figure 9a. Given the training set of formulas, and the use of the *Constrained* edge matching criterion, the start state is disconnected from all downstream states as no transition-centric options match with the edge transitions. Therefore, the agent does not attempt to solve the task and returns failure with the reason being *no feasible path*, i.e. a disconnected reward machine graph after removing infeasible edges.

If the *Relaxed* edge matching criterion is used, there are matching transition-centric options for each of the RM edges. The trajectory adopted by the agent when transferring the policies is depicted in

Figure 10. The agent collects all the three requisite resources before it terminates the task execution. Further note that the agent passes through a $wood$ resource grid as the specification does not explicitly prohibit it.



(a)



(b)

Figure 9: Figure 9a depicts the reward machine for the the specification $\varphi_{test} = \mathbf{F}workbench \wedge \mathbf{F}grass \wedge \mathbf{F}axe$. Figure 9b depicts the edges that do not have a compatible transition-centric option for the *Constrained* edge matching criterion. Note that all the edges have at least one matched transition-centric option for the *Relaxed* criterion.

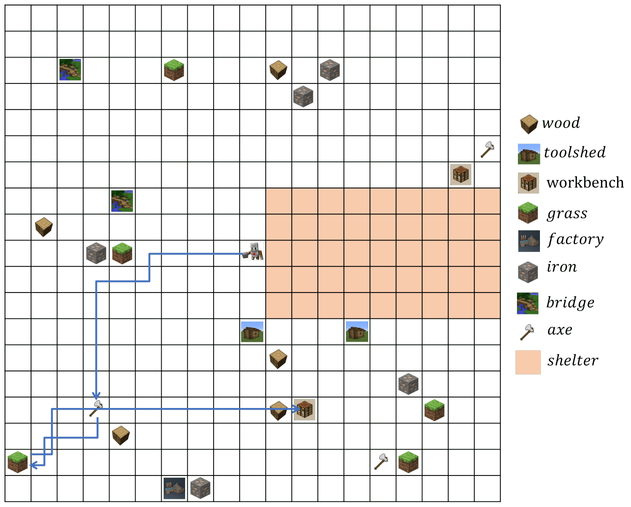


Figure 10: Trajectory executed by the agent using LTL-Transfer on the specification $\varphi_{test} = \mathbf{F}workbench \wedge \mathbf{F}grass \wedge \mathbf{F}axe$.

10.4 Example specifications

Here we provide the specifications and the interpretations of three formulas each from the *Hard*, *Soft*, *Strictly Soft*, *No Orders*, and *Mixed* formula types. Note that the training set contained 50 formulas each (new formulas were added incrementally when varying the training set size), and the test set contained 100 formulas each.

Hard: Example formulas belonging to the *Hard* dataset are as follows:

1. $\mathbf{F}wood \wedge \mathbf{F}axe \wedge \neg wood\ \mathbf{U}\ grass \wedge \neg grass\ \mathbf{U}\ workbench \wedge \neg workbench\ \mathbf{U}\ bridge$: Visit $bridge$, $workbench$, $grass$, $wood$, and axe. Ensure that $bridge$, $workbench$, $grass$, $wood$ in that particular order. Objects later in the sequence cannot be visited before the prior objects.
2. $\mathbf{F}workbench \wedge \mathbf{F}factory \wedge \mathbf{F}iron \wedge \mathbf{F}shelter \wedge \neg factory\ \mathbf{U}\ axe$: Visit $workbench$, $factory$, $iron$, $shelter$, and axe. Ensure that $factory$ is not visited before axe.
3. $\mathbf{F}toolshed \wedge \mathbf{F}bridge \wedge \mathbf{F}factory \wedge \mathbf{F}axe \wedge \neg bridge\ \mathbf{U}\ wood$: Visit $toolshed$, $bridge$, $factory$, axe, and $wood$. Ensure that $bridge$ is not visited before $wood$.

Soft: Examples belonging to the *Soft* dataset are as follows:

1. $\mathbf{F}(bridge \wedge \mathbf{F}(factory \wedge \mathbf{F}(iron \wedge \mathbf{F}shelter)))$: Visit $bridge$, $factory$, $iron$, and $shelter$ in that sequence. The objects later in the sequence may be visited before the prior objects, provided that they are visited at least once after the prior object has been visited.
2. $\mathbf{F}workbench \wedge \mathbf{F}(factory \wedge \mathbf{F}grass)$: Visit the $workbench$, $factory$, and $grass$: Visit $grass$ at least once after visiting the $factory$.
3. $\mathbf{F}(axe \wedge \mathbf{F}factory) \wedge \mathbf{F}workbench$: Visit axe, $factory$, and $workbench$. Ensure that $factory$ is visited at least once after axe is.

Strictly Soft: Examples belonging to the *Strictly Soft* dataset are identical to the *Soft* specifications, except they do not allow for simultaneous satisfaction of multiple sub-tasks. The subtasks in sequence must occur strictly temporally after the prior subtask. This is enforced using $\mathbf{XF}a$ instead of $\mathbf{F}a$.

No Orders: These specifications only contain a list of subtasks to be completed. No temporal orders are enforced between the various subtasks.

Mixed: Examples belonging to the *Mixed* dataset are as follows:

1. $\mathbf{F}toolshed \wedge \mathbf{F}factory \wedge \neg toolshed \ \mathbf{U} \ shelter \wedge \mathbf{F}(grass \wedge \mathbf{F}bridge)$: Visit the $toolshed$, $factory$, $shelter$, $grass$, and $bridge$. Ensure that $toolshed$ is not visited before the shelter, and $bridge$ is visited at least once after $grass$.
2. $\mathbf{F}grass \wedge \neg grass \ \mathbf{U} \ toolshed \wedge \mathbf{F}(factory \wedge \mathbf{XF}workbench)$: Visit $grass$, $toolshed$, $factory$, and $workbench$. Ensure that $grass$ is not visited before $toolshed$, and $workbench$ is at least visited once strictly after $factory$.
3. $\mathbf{F}iron \wedge \neg iron \ \mathbf{U} \ toolshed \wedge \mathbf{F}(shelter \wedge \mathbf{XF}wood)$: Visit $iron$, $toolshed$, $shelter$, and $wood$. Ensure that $iron$ is not visited before $toolshed$, and $wood$ is at least visited once strictly after $shelter$.

10.5 Robot demonstrations

We demonstrated LTL-Transfer on Spot [1], a quadruped mobile manipulator, in a household environment, as shown in Figure 11, where the robot can fetch and deliver objects while navigating through the space.

LTL-Transfer first trained policies to solve 2 training tasks $\Phi_{train} = \{\neg desk_a \ \mathbf{U} \ book \wedge \mathbf{F}desk_a, \neg desk_b \ \mathbf{U} \ juice \wedge \mathbf{F}desk_b\}$ in simulation. Then we demonstrated the zero-shot transfer capability of LTL-Transfer on a set of test tasks, as shown in Table 1. The robot can complete test tasks that it is expected to succeed, as shown in an example video [5]. For the test tasks that it is expected to fail, the robot as expected does not start execution because LTL-Transfer does not produce a feasible path through the reward machine graph given the transition-centric options learned from training tasks Φ_{train}.

The state space of the household environment includes the locations of the robot and the 4 objects, i.e. 2 desks, a book on a bookshelf and a juice bottle on a kitchen counter. The robot can move in 4 cardinal directions deterministically, and an invalid movement does not change the robot's position. The robot performs the pick action after it moves to the grid cell representing $book$ or $juice$. We finetuned an off-the-shelf object detection model [6] to determine the grasp point from an RGB image by selecting the center point of the most confident bounding box over the target object. The robot performs the place action after it moves to the grid cell representing $desk_a$ or $desk_b$ at the end of a trajectory by an option policy while carrying an object, i.e. a book or a juice bottle.

[5]Video: `https://youtu.be/FrY7CWgNMBk`

[6]`https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/`

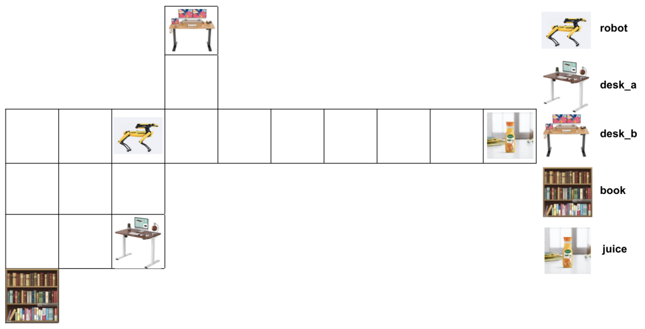


Figure 11: The map of the household environment and the corresponding propositions used for the robot demonstrations.

Table 1: 10 Test LTL Tasks for Robot Demonstrations.

Test LTL Task	**Expected to**
$\mathbf{F}(book \wedge \mathbf{F}(desk_a \wedge \mathbf{F}(juice \wedge \mathbf{F}desk_a)))$	succeed
$\mathbf{F}(book \wedge \mathbf{F}(desk_a \wedge \mathbf{F}(juice \wedge \mathbf{F}desk_b)))$	succeed
$\mathbf{F}(book \wedge \mathbf{F}(desk_b \wedge \mathbf{F}(juice \wedge \mathbf{F}desk_a)))$	succeed
$\mathbf{F}(book \wedge \mathbf{F}(desk_b \wedge \mathbf{F}(juice \wedge \mathbf{F}desk_b)))$	succeed
$\mathbf{F}(juice \wedge \mathbf{F}(desk_a \wedge \mathbf{F}(book \wedge \mathbf{F}desk_a)))$	succeed
$\mathbf{F}(juice \wedge \mathbf{F}(desk_a \wedge \mathbf{F}(book \wedge \mathbf{F}desk_b)))$	succeed
$\mathbf{F}(juice \wedge \mathbf{F}(desk_b \wedge \mathbf{F}(book \wedge \mathbf{F}desk_a)))$	succeed
$\mathbf{F}(juice \wedge \mathbf{F}(desk_b \wedge \mathbf{F}(book \wedge \mathbf{F}desk_b)))$	succeed
$\neg desk_a \ \mathbf{U} \ book \ \wedge \ \neg juice \ \mathbf{U} \ desk_a \ \wedge \ \neg desk_b \ \mathbf{U} \ juice \ \wedge \ \mathbf{F}desk_b$	fail
$\neg desk_b \ \mathbf{U} \ juice \ \wedge \ \neg book \ \mathbf{U} \ desk_b \ \wedge \ \neg desk_a \ \mathbf{U} \ book \ \wedge \ \mathbf{F}desk_a$	fail