LTL-Transfer: Skill Transfer for Temporal Task Specification

Jason Xinyu Liu\textsuperscript{*1}, Ankit Shah\textsuperscript{*1}, Eric Rosen\textsuperscript{1}, Mingxi Jia\textsuperscript{1}, George Konidaris\textsuperscript{1} and Stefanie Tellex\textsuperscript{1}

Abstract—Deploying robots in real-world environments, such as households and manufacturing lines, requires generalization across novel task specifications without violating safety constraints. Linear temporal logic (LTL) is a widely used task specification language with a compositional grammar that naturally induces commonalities among tasks while preserving safety guarantees. However, most prior work on reinforcement learning with LTL specifications treats every new task independently, thus requiring large amounts of training data to generalize. We propose LTL-Transfer, a zero-shot transfer algorithm that composes task-agnostic skills learned during training to safely satisfy a wide variety of novel LTL task specifications. Experiments in Minecraft-inspired domains show that after training on only 50 tasks, LTL-Transfer can solve over 90% of 100 challenging unseen tasks and 100% of 300 commonly used novel tasks without violating any safety constraints. We deployed LTL-Transfer at the task-planning level of a quadruped mobile manipulator to demonstrate its zero-shot transfer ability for fetch-and-deliver and navigation tasks.

I. INTRODUCTION

Deploying robots in the real world requires generalization across many novel tasks while preserving safety. For example, an industrial robot that fetches the same components in different orders based on the assembled product should only learn to fetch each part once. These tasks typically share constituents like objects and trajectory segments, which creates an opportunity to reuse knowledge [1].

Linear temporal logic (LTL) [2] is an effective means of specifying objectives, including safety constraints, for reinforcement learning (RL) agents [3], [4], [5]. Its compositional grammar reflects the compositional nature of most tasks. However, most prior approaches to RL for LTL tasks learn to solve every new task from scratch. We propose a zero-shot transfer algorithm, LTL-Transfer, that exploits the compositionality of LTL task specification to safely solve novel tasks without additional training by composing skills learned in prior tasks. Transferring skills is more data-efficient than learning from scratch and more computationally efficient than planning.

We show in a simulated Minecraft-inspired domain that LTL-Transfer can solve over 90% of 100 challenging unseen tasks and 100% of 300 common and less complex novel tasks after training on only 50 tasks and never violates a safety constraint. We deploy LTL-Transfer at the task-planning level of a quadruped mobile manipulator to solve fetch-and-deliver and navigation tasks in zero-shot. Our key insight is efficiently reusing learned skills by leveraging similarities between the novel and training tasks. Code, datasets, supplementary materials, and robot demonstration videos are at https://jasonxlyiu.github.io/LTL-Transfer

\textsuperscript{*}Equal contribution. \textsuperscript{1} Brown University.
translation of an LTL specification to a transition-based representation. We consider task specification in the co-safe fragment of LTL [11], [12] where formulas can be verified by a finite trace, thus making it ideal for episodic tasks. Camacho et al. [5] showed that every co-safe LTL formula \( \phi \) can be translated to an equivalent reward machine (RM) [13], [14].

Mathematically, the reward machine \( M_\phi \) is a 5-tuple \( (Q_\phi, q_0, Q_{term,\phi}, \varphi, T_\phi, R_\phi) \), where \( Q_\phi \) is the finite set of states, \( q_0, \) is the initial state, \( Q_{term,\phi} \) is the set of terminal states; \( T_\phi : Q_\phi \times 2^{[1]} \to Q_\phi \) is the deterministic transition function; and \( R_\phi : Q_\phi \to \mathbb{R} \) represents the reward accumulated by entering a given state. Figure 2 shows the reward machine graph representing the LTL formula \( F_{\text{wood}} \land \neg F_{\text{axe}} \). Nodes encode progressed RM states, 0 for an accepting state and 3 for a failure state. Boolean formulas label edges. Truth assignments of propositions \( \alpha \) that satisfy edges induce transitions in the RM. Our proposed algorithm, LTL-Transfer, for transferring learned policies to solve novel LTL specifications, is compatible with all algorithms that generate policies by solving a product MDP of the reward machine \( M_\phi \) and the task environment.

**Options Framework:** Sutton and Barto [15] introduced a framework, termed options or skills, for incorporating temporally extended actions into reinforcement learning. An option \( o = (I, \beta, \pi) \) is defined by the initiation set \( I \), a set of states where the option policy can be executed; the termination condition \( \beta \), which determines when the execution ends; and the option policy \( \pi \). Our proposed algorithm, LTL-Transfer, compiles policies learned during training into task-agnostic options that are transferred to solve novel tasks.

### III. RELATED WORK

Most approaches extending reinforcement learning (RL) to temporal tasks first generate a product MDP of the state space and the automaton representation of the LTL task specification [3], [14], [4], [5], [13]. Notably, Jothiramgulan et al. [16] proposed interleaving graph-based planning on the automaton with RL to bias exploration towards trajectories that satisfy the LTL specification. Although these approaches exploited the compositional structure of LTL to accelerate learning, they did not exploit the compositionality to transfer to novel tasks. Thus, the policy to satisfy a novel LTL specification must be learned from scratch.

A common approach towards generalization across temporal tasks has been to learn independent policies for subtasks [17], [18], [19], [20]. Given a new task, these methods then sequentially compose the learned policies in an admissible order. Consider the Minecraft-inspired grid world depicted in Figure 2 containing \( \text{wood} \) and \( \text{axe} \) objects. The subtask-based methods learn policies to solve subtasks \( F_{\text{wood}} \) and \( F_{\text{axe}} \) involving reaching each object. When tasked with the specification \( \varphi_{\text{test}} = F_{\text{wood}} \land \neg F_{\text{axe}} \), i.e., collect \( \text{wood} \) but do not collect \( \text{wood} \) until \( \text{axe} \) is collected, the subtask-based methods would violate the ordering constraint by reaching \( \text{axe} \) through the grid cells containing \( \text{wood} \). These approaches rely on additional fine-tuning to solve the target task correctly. We propose a general framework for transferring learned policies to novel tasks in zero-shot without violating any safety constraints.

Kuo et al. [21] proposed learning subnetworks for propositions and operators, then creating the final policy network through composition. Vaeypoor et al. [22] proposed learning a latent embedding over LTL formulas using a graph neural network to solve novel LTL tasks. In contrast, our method uses formal methods to identify learned policies best suited for transfer, thus requiring orders of magnitude fewer training tasks to achieve comparable results. Furthermore, neither method can guarantee the preservation of safety constraints like our approach. Xu et al. [23] considered transfer learning between pairs of source and target tasks, while our approach trains on a collection of tasks and transfers to a set of novel tasks. Nangue Tasse et al. [24] attempted zero-shot composition through logical and temporal compositions of policies, but their approach assumes that a given task must be satisfiable. By leveraging the structure of the task automaton, our proposed algorithm aborts execution immediately after it identifies the task as unsolvable given the learned skills.

A qualitatively distinct approach to zero-shot generalize to novel tasks is model-based RL that plans with an estimated transition model [25], [26]. Compared to model-based methods, our approach transfers learned policies in zero-shot thus requires significantly less computation at test time.

Our approach draws inspiration from prior works on learning portable skills in Markov domains [27], [28], [29], [30]. These approaches learn task-agnostic representations of preconditions, constraints, and effects of an option [15]. We learn portable skills to satisfy novel LTL task specifications.

### IV. PROBLEM DEFINITION

We represent the environment as an MDP without the reward function \( M_\Sigma = (S, A, T_\Sigma) \), where \( S \) is the set of states, \( A \) is the set of actions, and \( T_\Sigma : S \times A \times S \to [0, 1] \) represents the transition dynamics of the environment which we assume to be hidden from the RL algorithm. Further, a set \( \alpha \) of Boolean propositions represents the facts about the environment and forms the compositional building blocks for specifying tasks. We assume a labeling function \( L : S \to 2^{[1]} \) that maps the state to the Boolean propositions is given. A task in the environment \( M_\Sigma \) is specified by a linear temporal logic (LTL) formula \( \varphi \), and it is translated to a reward machine \( M_\varphi = (Q_\varphi, q_0, Q_{term,\varphi}, \varphi, T_\varphi, R_\varphi) \) detailed in Section II.

Given a set of training tasks \( \Phi_{\text{train}} = \{ \varphi_1, \varphi_2, \ldots, \varphi_n \} \), specified by LTL, and policies for a set of options \( O \) learned from these training tasks, a zero-shot transfer algorithm needs to solve a novel LTL task \( \varphi_{\text{test}} \not\in \Phi_{\text{train}} \) in the same environment without additional training.

### V. LTL-TRANSFER WITH TRANSITION-CENTRIC OPTIONS

#### A. Algorithm Overview

Consider the environment map depicted in Figure 2. Assume an RL algorithm has learned option policies to collect \( \text{axe} \) and \( \text{wood} \) individually from the training tasks \( F_{\text{axe}} \) and \( F_{\text{wood}} \), respectively. Now given the test task \( \varphi_1 = F(\text{axe} \land F_{\text{wood}}) \), i.e., first collect \( \text{axe} \), then \( \text{wood} \), a
transfer algorithm should identify that sequentially composing the policies for $\text{F\_ax}$ and $\text{F\_wood}$ solves the new task $\varphi_1$ (as depicted in blue). Then consider a different test task $\varphi_2 = \text{F\_wood} \land \neg\text{wood} \Rightarrow \text{ax}$, i.e., first collect $\text{axe}$, then $\text{wood}$, but avoid $\text{wood}$ until $\text{axe}$ is collected. The transfer algorithm must realize that the policy that satisfies $\text{F\_axe}$ does not guarantee avoiding $\text{wood}$ while going to $\text{axe}$. Therefore, it must not start execution using only these two learned policies to avoid accidentally violating the ordering constraint.

We developed a zero-shot transfer algorithm, LTL-Transfer, that composes learned policies to solve novel LTL tasks while enforcing ordering constraints. It operates in two stages.

1) First, LTL-Transfer accepts the set of training tasks $\Phi_{\text{train}}$ and the policies learned from the training tasks and compiles a set of task-agnostic, portable options $O_e$.

2) Next, it identifies and executes a sequence of options from the set $O_e$ to solve a novel task $\varphi_{\text{test}}$.

We can use a class of RL algorithms that operate on a product MDP composed of the environment $\mathcal{M}_e$ and the reward machine $\mathcal{M}_r$ to learn option policies from the training LTL tasks [3], [14], [4], [5], [13]. We chose LPOPL [4] because it explicitly allows for sharing policies across LTL task specifications that share progression states. Given a set of LTL tasks, LPOPL first translates each task to a reward machine (RM) and decomposes tasks to subtasks, each of which corresponds to a state in the RM, then learns an action-value function, represented by a DQN [31], for each subtask. Please see supplementary materials for implementation details of LPOPL. The learned policy is Markov with respect to the environment states $S$ for a given RM state, i.e., the policy to be executed in the RM state $q \in Q_e$ is $\pi^e_q : S \rightarrow A$.

Executing the state-centric option $o^c_q \in O_q$ with the policy $\pi^e_q$ from the reward machine state $q$ triggers a transition in the RM on a path to an acceptance state. There can be multiple outgoing transitions from an RM state, so the target RM transition of an option $o^c_q$ is conditioned on the environment state $s \in S$ where the execution was initiated. We propose compiling each state-centric option, $o^c_q$, into multiple transition-centric options by partitioning the initiation set of the state-centric option based on the estimated success probability of its policy satisfying the target RM transition from the starting environment state. Each resulting transition-centric option will maintain the satisfaction of self-transition edge $e^r_{q,q}$ from the starting RM state $q$ until it triggers the target RM transition $e^r_{q,q'}$. Our insight is that each transition-centric option triggers a transition in the reward machine on a path to an acceptance state, and these RM transitions may be shared across different tasks. Thus, the transition-centric options $O_e$ are portable across different tasks. We describe the compilation algorithm in Section V-B.

Given a novel LTL task specification $\varphi_{\text{test}} \notin \Phi_{\text{train}}$, our transfer algorithm first constructs a reward machine representing the task, $\mathcal{M}_{\varphi_{\text{test}}}$, then identifies a path through the reward machine that can be traversed by sequentially executing transition-centric options from the set $O_e$. Our transfer algorithm is sound, and it terminates. We describe the details of the transfer algorithm in Section V-C.

The key advantage of our approach is that the option compilation can be done offline for any environment. We can then transfer the options to solve novel tasks at execution time in zero-shot. Thus, learning to solve a limited number of LTL tasks can help solve a wide gamut of unseen tasks.

B. Compilation of Transition-Centric Options

The policy learned to satisfy an LTL task specification $\varphi$ identifies the current reward machine state $q \in Q_e$ and executes a Markov policy $\pi^e_q$ until the state of the reward machine progresses. We represent this policy as a state-centric option, $o^c_q = (S, \beta^e_{q,q}, \pi^e_q)$, where the initiation set is the entire state space $S$ of the environment; the option terminates when the truth assignment of the propositions $\alpha$ violates the self-transition $e^r_{q,q}$ from the RM state $q$. The termination condition $\beta^e_{q,q}$ is formally defined as follows,

$$\beta^e_{q,q} = \begin{cases} 1, & \text{if } L(s) \neq e^r_{q,q} \\ 0, & \text{otherwise}. \end{cases}$$

A transition-centric option $o^t_{q,q',e^r_{q,q'}}$ executes a Markov policy that ensures that the truth assignment of the propositions $\alpha$ satisfies the self-transition edge $e^r_{q,q'}$ at all time until the policy yields a truth assignment that satisfies the target outgoing edge $e^r_{q,q'}$. We define a transition-centric option as follows,

$$o^t_{q,q',e^r_{q,q'}} = (S, \beta^t_{q,q'}, \pi^e_q, e^r_{q,q'}, f_{e^r_{q,q'}}).$$
The initiation set is the entire state space $S$ of the environment; the termination condition $\beta_{q,q}$ is defined by the violation of the self-transition edge $e_{q,q}$; the option’s policy is Markov $\pi_{q,q} : S \rightarrow A$; $e_{q,q}$ and $e_{q,q}'$ represent the self-transition and the target outgoing edge, respectively; and $f_{q,q}$ : $S \rightarrow [0,1]$ represents the success probability of the policy $\pi_{q,q}$ satisfying the target edge $e_{q,q}$ starting from the environment state $s \in S$.

Algorithm 1 describes our approach to compiling each state-centric option $\mathcal{O}_q$ to a set of transition-options $\{\sigma_{q,q} e_{q,q} : q \in \mathcal{Q}_q, q \text{’is out-neighbor of } q\}$. Executing the policy $\pi_{q,q}$ of the state-centric option may satisfy different outgoing edges $e_{q,q}^*$ of the reward machine state $q$ depending on what environment state $s \in S$ the execution was initiated. Thus, the success probability $f_{q,q}$ acts as a soft segmenter of the state space $S$; it can be estimated by policy rollouts from all environment states in discrete domains or using sampling-based methods [29], [30] in continuous domains.

C. Transferring to Novel LTL Task Specification

Algorithm 2 describes the zero-shot transfer algorithm that composes transition-centric options from the set $\mathcal{O}_e$ to solve a novel test task $\varphi_{test}$ in the environment $M_\mathcal{S}$. Line 2 generates the reward machine (RM) graph for the test task. Line 3 examines each edge $e_{q,q}$ of the RM, identifies the transition-centric options that satisfy that edge transition, and prunes an edge if no such option is found. Line 4 identifies and caches all paths from the current state to the accepting state $q$ of the reward machine. Lines 8 and 9 identify a set of all eligible options that can potentially achieve an RM transition from the current state to a progressed state on a path to the accepting state.

Lines 12 and 13 then execute the option with the highest success probability of satisfying the target edge transition, estimated by $f$. If the option fails to progress to another RM state, we delete it from the set (Line 15) and execute the next option. If the set of eligible options is empty at any point without reaching the accepting RM state $q^T$, Line 17 exits with a failure. A successful transfer occurs when the RM progresses to the accepting state $q^T$.

D. Matching Options to Reward Machine Transitions

The edge-matching conditions determine whether we can safely apply a transition-centric option to transition along an edge of the reward machine (RM). We used the edge-matching conditions to prune the RM graph to retain only the edges matched with available options (Algorithm 2 Line 3) and identify eligible options from a given RM state (Algorithm 2 Line 8). Given a test task $\varphi_{test}$, when the current RM state is $q$, its self-transition edge is $e_{q,q}$ and the target outgoing edge is $e_{q,q}^*$, to determine if a transition-centric option $\sigma_{e_1,e_2}$ matches the target transition in the RM, we propose two edge-matching conditions, Constrained and Relaxed. Both ensure the task execution does not fail due to an unsafe transition.

Constrained Edge-Matching Condition requires that every truth assignment satisfying the self-transition edge $e_1$ of the option also satisfies the self-transition edge $e_{q,q}$ of the reward machine $M_{\varphi_{test}}$. Similarly, every truth assignment satisfying the outgoing edge $e_2$ of the option must satisfy the target transition $e_{q,q}$ of the test task’s RM. This strict requirement reduces the applicability of the learned options but ensures that the target edge is always achieved.

Constrained edge-matching condition is satisfied if the following Boolean expression holds:

$$\neg sat(e_1 \land \neg e_{q,q}^*) \land \neg sat(e_2 \land \neg e_{q,q}^*)$$.

Let $sat(g)$ be a Boolean function that is true if and only if a truth assignment exists to satisfy the Boolean formula $g$.

Relaxed Edge-Matching Condition requires the self edges $e_1$ and $e_{q,q}^*$ share satisfying truth assignments, so must the outgoing edges $e_2$ and $e_{q,q}^*$. However, it allows the option to have valid truth assignments that may not satisfy the target outgoing edge, yet none of the truth assignments should trigger a transition to an unrecoverable failure state $q^\perp$. We identify $q^\perp$ as a sink state of the RM graph without outgoing edges. Further, all truth assignments that terminate the option must not satisfy the self-transition edge of the test task’s reward machine. The Relaxed edge-matching condition can

![Algorithm 1 Compile State-Centric Options to Transition-Centric Options](image)

![Algorithm 2 Zero-shot transfer to test task $\varphi_{test}$](image)
We conducted experiments on four maps, similar to that We implemented the Relaxed edge-matching condition since there is no shared memory requirements. We implemented the Relaxed edge-matching condition using a propositional model counting algorithm [32] from SymPy [33], and the Constrained edge-matching condition using string comparison to circumvent the model counting problem and found a significant speed-up.

VI. EXPERIMENTS

The aim of our evaluation is to test the hypothesis that LTL-Transfer can efficiently transfer learned skills to solve novel temporal tasks while preserving safety guarantees. We tested our hypothesis in simulation and on a physical robot. The simulation domain allows us to scale skill transfer across a wide variety of tasks, while the real robot demonstrates the practicality of our approach for real-world mobile manipulation tasks. We first defined five types of specifications of varying complexity for training and testing, then conducted experiments to evaluate the following hypotheses,

1) **H1**: LTL-Transfer exceeds the baselines’ success rates of solving novel tasks.

2) **H2**: Relaxed edge-matching condition leads to higher success rates than the Constrained condition.

3) **H3**: Transferring learned policies to certain specification types (introduced in Section VI-B) leads to higher success rates.

4) **H4**: LTL-Transfer is robust to highly uncertain transition dynamics.

A. Task Environment

We tested LTL-Transfer in a Minecraft-inspired grid-world domain commonly seen in compositional reinforcement learning (RL) and integration of temporal logics with RL [20], [4], [16], [19]. This domain is particularly well suited for testing transfer learning with temporal tasks as policies to solve individual tasks can be learned rapidly with few computational resources. Thus, we can run comprehensive evaluations of skill transfer across a full factorial of training and test task types that cover a wide variety of LTL templates. We conducted experiments on four maps, similar to that depicted in Figure [2] with a dimension of 19 × 19. Each location in the map is either vacant or occupied by one of nine object types. Multiple instances of an object type may occur across the map. After the agent enters a grid cell occupied by an object, the proposition representing that object type becomes true. Actions are moving in four cardinal directions; an invalid action results in no movement. We tested zero-shot transfer in both deterministic and stochastic domains.

B. Types of Task Specifications

For a comprehensive evaluation of transferring learned policies across various LTL specifications, we considered the following three types of ordering constraints, proposed by Shah et al. [8], which constitute five specification types. Each constraint is defined on a pair of propositions a and b. Without loss of generality, we assume that a precedes b.

1) **Hard** ordering constraints occur when b must never be true before a. This property is expressed through the LTL formula ¬b U a.

2) **Soft** ordering constraints allow b to occur before a as long as b happens at least once after a becomes true for the first time. Soft orders are expressed through the LTL formula F(a ∧ Fb).

3) **Strictly Soft** ordering constraints are similar to soft orders except that 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 through the LTL formula F(a ∧ XFb).

We sampled five training sets, with 50 formulas in each, that represent five different specification types, Hard, Soft, Strictly-Soft, No-Orders, and Mixed, and a test set of 100 formulas for each type. This imitates the real-world scenario where users do not know the test task beforehand, so the robot must be trained on a limited set of training tasks then transfer to a wide variety of novel tasks. When constructing a test set, we excluded tasks already in the training set. All specifications in the Hard, Soft, and Strictly-Soft sets were expressed using the respective templates described above. No-Orders sets have no ordering constraints, e.g., F(a ∧ Fb ∧ Fc). In the Mixed set, each binary precedence constraint was expressed as one of the three ordering constraints described above. A given task in the environment involves visiting a specified set of object types in an admissible order determined by ordering constraints. Please see supplementary materials for example formulas from each specification type.

C. Results and Discussion

Comparison with Baselines: We compare the performance of LTL-Transfer to three baselines, i.e., LTL2Action [22], LPOPL [4], and a random policy.

**LTL2Action** proposed by Vaezipoor et al. [22] embeds LTL specifications using a graph neural network and sequentially selects the next proposition to satisfy. This pre-trained embedding is appended to the state features to yield a goal-conditioned task policy, termed LTL Bootcamp, which serves as the upper bound of LTL2Action’s performance on novel tasks. We trained the LTL Bootcamp on the same training sets as LTL-Transfer and compared their performance.

**LPOPL**, detailed in Section V-A, serves as the lower bound that any transfer algorithm must surpass due to its limited transfer ability to solve novel tasks. While LPOPL was not explicitly designed for zero-shot transfer, it can satisfy task specifications that are a progression of a training LTL formula because of its use of LTL progression and multi-task learning.

\[
\begin{align*}
\text{VI-B:} & \quad \text{H1: LTL-Transfer exceeds the baselines’ success rates of solving novel tasks.} \\
& \text{H2: Relaxed edge-matching condition leads to higher} \\
& \text{success rates than the Constrained condition.} \\
& \text{H3: Transferring learned policies to certain specification} \\
& \text{types (introduced in Section VI-B) leads to higher} \\
& \text{success rates.} \\
& \text{H4: LTL-Transfer is robust to highly uncertain transition} \\
& \text{dynamics.}
\end{align*}
\]
Figure 3a depicts the success rate of all the methods on 100 novel tasks in each test set described in Section VI-B after training on 50 tasks of the Mixed type. LPOPL performed worse than the random policy as it did not attempt to satisfy any specification that did not exist in its progression set. Both LTL-Transfer and LTL2Action have near-perfect success rates on Soft, Strictly-Soft, and No-Orders test set as these specifications do not have irrecoverable failure state. Specifications in Hard and Mixed test sets have failure states. Thus, we observe a lower success rate across all methods. However, LTL-Transfer demonstrates the best transfer success rate in the difficult test sets. Crucially, Figure 3b shows that LTL-Transfer never violated any specification, while LTL2Action’s specification violation rate is similar to that of the random policy, which could mean that LTL2Action essentially acts randomly given an infeasible task.

Effect of Edge-Matching Condition: We trained LTL-Transfer on Mixed training sets of varying sizes and tested zero-shot transfer on all five test sets. The success rates of using the Relaxed and Constrained edge-matching conditions are depicted in Figure 3d and Figure 3c respectively. We observed that LTL-Transfer using the Relaxed edge-matching condition successfully transferred to significantly more novel specifications across all types, thus supporting H2.

Relative Difficulty of Specification Types: Figure 3c shows that LTL-Transfer with the Relaxed edge-matching condition can perfectly transfer to novel tasks of Soft, Strictly-Soft and No-Orders types after training on very few tasks. After training on 50 tasks, LTL-Transfer can transfer to over 95% of novel tasks of the Mixed type. Tasks of the Hard type are the most difficult to transfer to. Figure 3d shows that the different tasks are equally difficult to transfer to using the Constrained edge-matching condition. Thus, H3 is supported only by using the Relaxed edge-matching condition.

Stochastic Environments: To evaluate the robustness of LTL-Transfer to stochastic transitions, we increased the slip probability from 0 to 0.4 and observed a slight 3% decrease in transfer success in Figure 3e, which supports H4.

D. Robot Demonstrations

We deployed LTL-Transfer at the task-planning level of a quadruped mobile manipulator, Spot [34], with off-the-shelf motion planning and grasping capabilities in a discretized indoor environment where the robot can fetch and deliver objects while navigating through the space. LTL-Transfer first learned policies from 20 training tasks, then transferred the learned skills to 100 novel tasks from the five specification types defined in Section VI-B. 50 of which were executed on the robot. The environment contains 31 grid cells, but LTL-Transfer works in larger domains, like 19 × 19, as presented in Section VI-A. The state space includes the locations of the robot and six landmarks, i.e., two desks, a couch, a door, a bookshelf with a book, and a kitchen counter with a juice bottle on top. Actions are moving in the four cardinal directions. Only navigation skills are learned from the training tasks. Picking is executed when the goal location of a navigation option is the bookshelf or the counter. Placement is executed after the robot reaches the desk or the couch and has picked up an object.

VII. CONCLUSIONS

We introduced LTL-Transfer, a zero-shot transfer algorithm that uses the compositionality of LTL task specification to maximally transfer learned policies to solve various novel tasks. Experiments in deterministic and stochastic Minecraft-inspired domains showed that LTL-Transfer can solve complex unseen tasks without violating any safety constraints. We deployed LTL-Transfer at the task-planning level of a mobile manipulator to safely solve fetch-and-deliver and navigation tasks in zero-shot. We envision incorporating long-term planning and intra-option policy updates to produce not just satisfying but optimal solutions to novel tasks.

ACKNOWLEDGMENT

The authors thank Peilin Yu and Mingxi Jia for editing the videos. This work is supported by ONR under grant numbers N00014-21-1-2584, N00014-17-1-2699, and N00014-22-1-2592, NSF under grant number CNS-2038897, Amazon Robotics under award number 1061079, and funding from Echo Labs. Partial funding for this work was provided by The Boston Dynamics AI Institute (“The AI Institute”).

Videos are at https://jasonxyliu.github.io/
A DQN policy is a feedforward network with two hidden layers, each of which has 64 ReLU units. We used the same hyperparameters suggested in [4] for training, i.e., the learning rate is 0.0001; the size of the replay buffer is 25,000; 32 transitions are randomly sampled from the replay buffer for every update; the discount factor is 0.9; exploration decreases linearly from 1 to 0.02.

APPENDIX II
EXAMPLE LTL FORMULAS

We provide LTL task specifications and their interpretations from the Hard, Soft, Strictly Soft, No Orders, and Mixed formula types.

**Hard:** Example formulas and their interpretations from the Hard type are as follows:

1) \( F_{\text{workbench}} \land F_{\text{factory}} \land F_{\text{iron}} \land F_{\text{shelter}} \land \neg F_{\text{factory}} U \text{ axe: Visit workbench, factory, iron, shelter, and axe. Ensure that factory is not visited before axe.} \)

2) \( F_{\text{toolshed}} \land F_{\text{bridge}} \land F_{\text{factory}} \land F_{\text{axe}} \land \neg F_{\text{bridge}} U \text{ wood: Visit toolshed, bridge, factory, axe, and wood. Ensure that bridge is not visited before wood.} \)

3) \( F_{\text{wood}} \land F_{\text{axe}} \land \neg F_{\text{wood}} U \text{ grass } \land \neg F_{\text{grass}} U \text{ workbench } \land \neg F_{\text{workbench}} U \text{ bridge: Visit bridge, workbench, grass, wood, and axe. Ensure visiting bridge, workbench, grass, and wood in that particular order. Objects that occur later in the sequence cannot be visited before any prior objects.} \)

**Soft:** Example formulas and their interpretations from the Soft type are as follows:

1) \( F(bridge) \land F(factory) \land F(iron) \land F(shelter)): Visit bridge, factory, iron, and shelter in that sequence. The objects that occur 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) \( F_{\text{workbench}} \land F(factory) \land F(grass): Visit the workbench, factory, and grass: Visit grass at least once after visiting the factory. \)

3) \( F(\text{axe}) \land F(factory) \land F_{\text{workbench}}: Visit axe, factory, and workbench. Ensure that factory is visited at least once after axe. \)

**Strictly Soft:** Example formulas and their interpretations from the Strictly Soft type are identical to the Soft specifications, except they do not allow simultaneous satisfaction of multiple sub-tasks. The subtasks in the sequence must occur strictly after the prior subtask. This is enforced using nested operators next and finally \( XF \) instead of \( F \).

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

1) \( F_{\text{wood}} \land F_{\text{grass}} \land F_{\text{stone}:} Visit bridge: Collect wood, grass, stone in no particular order.

2) \( F_{\text{toolshed}} \land F_{\text{factory}} \land \neg F_{\text{toolshed}} U \text{ shelter } \land F_{\text{grass}} \land F_{\text{bridge}}: Visit the toolshed, factory, shelter, grass, and bridge. Ensure that toolshed is not visited before the shelter and bridge is visited at least once after grass. \)

3) \( F_{\text{grass}} \land \neg F_{\text{grass}} U \text{ toolshed } \land F_{\text{factory}} \land X F_{\text{workbench}}: Visit grass, toolshed, factory, and workbench. Ensure that grass is not visited before toolshed and workbench is visited at least once strictly after factory. \)

**Mixed:** Example formulas and their interpretations from the Mixed type are as follows:

1) \( F_{\text{toolshed}} \land F_{\text{factory}} \land \neg F_{\text{toolshed}} U \text{ shelter } \land F_{\text{grass}} \land F_{\text{bridge}}: Visit the toolshed, factory, shelter, grass, and bridge. Ensure that toolshed is not visited before the shelter and bridge is visited at least once after grass. \)

2) \( F_{\text{grass}} \land \neg F_{\text{grass}} U \text{ toolshed } \land F_{\text{factory}} \land X F_{\text{workbench}}: Visit grass, toolshed, factory, and workbench. Ensure that grass is not visited before toolshed and workbench is visited at least once strictly after factory. \)

APPENDIX III
ADDITIONAL EXPERIMENTAL RESULTS

**Learning Curves for Various Training Sets:** We present learning curve trends of success rates for transferring policies learned on different specification types.

The learning curves for training on LTL tasks from the Hard training set with both Relaxed and Constrained edge-matching conditions are depicted in Figure 1a.

The learning curves for training on LTL tasks from the Soft training set with both Relaxed and Constrained edge-matching conditions are depicted in Figure 1b.

The learning curves for training on LTL tasks from the Strictly Soft training set with both Relaxed and Constrained edge-matching conditions are depicted in Figure 1c.

The learning curves for training on LTL tasks 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. We will include the plots in the final version of the paper.

![Fig. 1: Figure 1a depicts the success rates of transferring to five different specifications types using the Relaxed edge-matching condition after LTL-Transfer being trained on Hard training sets of various sizes. Figure 1b depicts the success rates with the Constrained edge-matching condition. Note that the error bars depict the 95% credible interval if the successful transfer was modeled as a Bernoulli distribution.](image)

Note that for training on each specification type, the learning curve trends are nearly identical to the learning
when LTL-Transfer is trained and tested on mixed specifications types using the Relaxed edge-matching condition after LTL-Transfer being trained on Soft training sets of various sizes. Figure 2b depicts the success rates with the Constrained edge-matching condition. Note that the error bars depict the 95% credible interval if the successful transfer was modeled as a Bernoulli distribution.

Curves of training on the Mixed specification types, as depicted in Figure 3 in the main paper. Hard specification types remain the most challenging specification types to transfer to.

Failure Analysis: As described in the main paper, we logged the reason for the failure of each unsuccessful transfer attempt. There are three possible causes:

1. Specification failure: a constraint is violated during execution, and the reward machine progresses to an unrecoverable state.
2. No feasible path: there are no paths connecting the start reward machine state to an accepting reward machine state with matching transition-centric options.
3. Options exhausted: there are no further transition-centric options available to further progress the state of the reward machine.

Figure 3 depicts the relative frequency of the failure modes when LTL-Transfer is trained and tested with mixed task specifications. Note that with the Relaxed edge-matching condition not progressing the task after utilizing all available safe options is the primary reason for failure (Figure 4a), whereas, with the Constrained edge-matching condition, the absence of feasible paths connecting the start and the accepting state is the primary reason for failure (Figure 4b).

Appendix IV

Selected Solution Trajectories in Simulation
Consider the case with Mixed training set with 5 formulas on Map 0. The training formulas are:

- $F_{\text{grass}} \land F_{\text{shelter}} \land F(\text{wood} \land X F_{\text{workbench}})$
- $F_{\text{toolshed}} \land F_{\text{workbench}} \land F_{\text{shelter}} \land (\neg\text{toolshed} \cup \text{shelter}) \land F(\text{grass} \land F_{\text{bridge}})$
- $F_{\text{toolshed}} \land F(\text{shelter} \land F(\text{axe} \land F_{\text{wood}}))$
- $F_{\text{iron}} \land F(\text{shelter} \land X F(\text{bridge} \land X F_{\text{factory}}))$
- $F_{\text{factory}}$

One of the Mixed test formulas is $\varphi_{\text{test}} = F_{\text{workbench}} \land F_{\text{grass}} \land F_{\text{axe}}$. The reward machine for this task specification is depicted in Figure 5a. Given the training set of formulas and the use of the Constrained edge-matching condition, the start reward machine state is disconnected from all downstream states as no transition-centric options match the edge transitions. Therefore, LTL-Transfer 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 condition is used, there are matching transition-centric options for each edge. The trajectory adopted by LTL-Transfer when transferring the policies is depicted in Figure 5b. The robot collects all three requisite resources before it terminates the task execution. Further, note that the robot passes through a grid containing wood as the specification does not explicitly prohibit it.

Appendix V

Robot Demonstration
The 50 test tasks executed on the robot are shown in Table II. The Proposition $a$ represents a brown desk, $b$ represents a white desk, $c$ represents a couch, $d$ represents a door, $s$ represents a bookshelf, and $k$ represents a kitchen counter.
Fig. 5: Figure 5a depicts the reward machine for the task specification $\varphi_{\text{test}} = F_{\text{workbench}} \land F_{\text{grass}} \land F_{\text{axe}}$, as well as all feasible edges matched by the Relaxed condition. Note that all the edges have at least one matching transition-centric option for the Relaxed edge-matching condition. Figure 5b depicts the edges that do not have a compatible transition-centric option for the Constrained edge-matching condition.

Fig. 6: Trajectory executed by the robot using LTL-Transfer to achieve the novel task specification $\varphi_{\text{test}} = F_{\text{workbench}} \land F_{\text{grass}} \land F_{\text{axe}}$. 
<table>
<thead>
<tr>
<th>LTL Task Specification</th>
<th>Type of Task</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. F_a</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>1. F_a ∧ F_b</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>2. F_a ∧ F_b ∧ F_C</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>3. F_a ∧ F_b ∧ F_s</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>4. F_a ∧ F_b ∧ F_k</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>5. F_a ∧ F_b ∧ F_C ∧ F_d</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>6. F_a ∧ F_b ∧ F_C ∧ F_s</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>7. F_a ∧ F_b ∧ F_C ∧ F_k</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>8. F_a ∧ F_b ∧ F_C ∧ F_k ∧ F_s</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>9. F(b ∧ F(a ∧ F(c ∧ F(d))))</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>10. F(s ∧ F_a)</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>11. F(s ∧ F_b)</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>12. F(a ∧ F_b)</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>13. F(b ∧ F_a)</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>14. F(a ∧ F(s ∧ F_c))</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>15. F(b ∧ F(s ∧ F_c))</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>16. F(s ∧ F(a ∧ F_c))</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>17. F(a ∧ F(b ∧ F_c))</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>18. F(s ∧ F(a ∧ F(k ∧ F_a)))</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>19. F(a ∧ F(b ∧ F(c ∧ F(d))))</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>20. F(s ∧ XF_a)</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>21. F(b ∧ XF_b)</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>22. F(a ∧ XF_b)</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>23. F(b ∧ XF_a)</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>24. F(a ∧ XF(b ∧ XF_c))</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>25. F(a ∧ XF(s ∧ XF_b))</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>26. F(b ∧ XF(s ∧ XF_a))</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>27. F(s ∧ XF(b ∧ XF_a))</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>28. F(k ∧ XF_b)</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>29. F(k ∧ XF_a)</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>30. ¬sUs ∧ Fs</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>31. ¬sUs ∧ F_b</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>32. ¬sUb ∧ Fs</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>33. bUk ∧ ¬sUb ∧ FC</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>34. bUk ∧ Fb</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>35. ¬sUs ∧ F_b</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>36. ¬sUs ∧ ¬bUs ∧ F_b</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>37. ¬sUs ∧ ¬bUs ∧ F_b</td>
<td>fetch and deliver</td>
<td>success</td>
</tr>
<tr>
<td>38. ¬sUs ∧ ¬sUb ∧ Fs</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>39. ¬sUb ∧ ¬sUs ∧ Fs</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>40. F_a ∧ F(b ∧ F_c)</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>41. F_a ∧ ¬sUb ∧ FC</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>42. F(a ∧ Fb) ∧ ¬sUs ∧ Fc</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>43. F_a ∧ F(b ∧ XF_c)</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>44. F(a ∧ Fb) ∧ ¬sUb ∧ Fc</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>45. F(b ∧ F_a) ∧ ¬sUb ∧ Fc</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>46. Fc ∧ ¬sUs ∧ Fs</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>47. Fc ∧ ¬sUs ∧ F_a</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>48. Fc ∧ ¬sUb ∧ Fs</td>
<td>navigation</td>
<td>success</td>
</tr>
<tr>
<td>49. Fc ∧ ¬sUb ∧ F_b</td>
<td>navigation</td>
<td>success</td>
</tr>
</tbody>
</table>