

APPENDIX

A LIMITATIONS

We note that our approach differs from standard approaches to determining a system, such as subset construction (Hopcroft et al., 2001). Those methods track the multiple states that could be reached under a given action. We wish to find a representation such that executing a policy corresponds exactly with transitioning from one region to another. We note in Remark 1 that a limiting factor of our approach is the agent does not necessarily take the shortest path in the environment due to the zero-shot nature of our solution. An extension to this work could investigate encoding region proximity into the TS or other methods for evaluating the choice of transitions.

RM’s approach is less conservative than our approach, so while they may collect additional symbols along the execution, they may be able to satisfy specifications on an environment with non-determinism, whereas we only provide deterministic solutions.

The reliance on deterministic MDPs is required for our theoretical guarantees, but it can be relaxed in practice. Another limitation is the sparse reward structure, which is shared with Nangue Tasse et al. (2020) and Leahy et al. (2024), and could be amended with reward shaping. The penalties for safety semantics caused the policies to have difficulty converging, especially when the environment is sparse. This is expected and, depending on environment complexity, a global reasoning to find a MV path may be required. We will add the above paragraph to the Limitations section of the manuscript.

B RELATED WORK

Our work lies at the intersection of temporal logic planning, reinforcement learning, and compositional methods for skill reuse. We review relevant literature in these areas and highlight how our approach differs from existing methods.

B.1 MODEL-BASED PLANNING FOR LINEAR TEMPORAL LOGIC

Linear Temporal Logic (LTL) has been widely used as a formal specification language for robot planning and control tasks due to its expressiveness and formal semantics (Belta et al., 2017). Traditional approaches to LTL planning rely on model-based techniques that synthesize controllers from formal specifications (Kress-Gazit et al., 2018). These methods typically construct product automata from LTL formulas and system models, then solve for satisfying trajectories through the product space (Kloetzer & Belta, 2008). More recent work has explored optimization-based approaches (Belta & Sadraddini, 2019), such as temporal logic motion planning using mixed-integer linear programming (Raman et al., 2014) or using graphs of convex sets (Kurtz & Lin, 2023), which can efficiently handle continuous state spaces and geometric constraints. While model-based planning methods provide formal correctness guarantees, they require accurate system models, including dynamics models, and can struggle with scalability in high-dimensional or uncertain environments. In contrast, our approach leverages learned policies that can handle model uncertainty, learn environment dynamics, and scale to complex domains, while still respecting the structure imposed by LTL specifications.

B.2 REINFORCEMENT LEARNING FOR LTL SPECIFICATIONS

Increasingly, research efforts have focused on adapting learning-based methods to satisfy temporal logic specifications. Early work in this direction explored using LTL formulas to shape reward functions (Li et al., 2016), enabling agents to learn behaviors that satisfy temporal properties, as well as methods for adapting Q-learning for signal temporal logic (STL) specifications (Aksaray et al., 2016).

A significant advancement came with the introduction of reward machines (Icarte et al., 2018), which provide a structured representation of tasks as finite state machines derived from LTL specifications. Reward machines expose the task structure to the learner, enabling more efficient learning through automatic decomposition. Building on this foundation, Toro Icarte et al. (2018) demonstrated how

a single RL agent could learn to satisfy multiple LTL specifications by leveraging shared structure across tasks.

Recent work has extended these ideas in several directions. Li et al. (2019) developed a formal methods approach to interpretable RL that maintains guarantees on learned behaviors. Cai et al. (2023) addressed the challenging problem of learning control policies for infeasible LTL specifications, seeking minimally-violating behaviors when satisfaction is impossible. Vaezipoor et al. (2021) proposed LTL2Action, which enables generalization across different LTL instructions in multi-task RL settings through learned embeddings of temporal specifications.

More recent approaches have explored the integration of LTL with goal-conditioned RL. León et al. (2022) introduced a method for following temporal specifications using latent goal representations, while Qiu et al. (2023) developed techniques for instructing goal-conditioned agents with temporal logic objectives. Jackermeier & Abate (2025) proposed DeepLTL, which learns to efficiently satisfy complex LTL specifications across multiple tasks.

While these RL-based approaches have shown impressive results, they typically require training separate policies for each new specification or compositional structure. Our work differs by enabling zero-shot composition of pre-trained policies to satisfy novel LTL specifications, eliminating the need for retraining when faced with new temporal objectives.

B.3 TRANSFER AND GOAL-CONDITIONED REINFORCEMENT LEARNING

Goal-conditioned RL extends standard RL by conditioning policies and value functions on desired goals, enabling generalization across objectives. In the context of logic and temporal logic specifications, several works have explored compositional methods. (Nangue Tasse et al., 2020) introduced a Boolean task algebra for RL that enables composition of learned policies through logical operators. More recently, the same authors proposed Skill Machines Nangue Tasse et al. (2024), which combine learned skills with automata-based task specifications to enable flexible policy composition.

(Jothimurugan et al., 2021) developed a compositional RL framework that learns policies for logical specifications by decomposing them into simpler sub-tasks on an automaton-like structure, learning policies corresponding to each edge, then finding a path through the automaton. While this approach is compositional, it is not zero-shot. (Liu et al., 2024) developed a similar approach, encoding specifications as an automaton and learning policies for executing edges in the automaton. When new automaton contain the same edges, those policies are reused. If a specification results in an automaton with previously unseen edges, new training is required.

While goal-conditioned RL typically focuses on reaching spatial goals, our work bridges the gap to temporal specifications by developing a framework for zero-shot composition of goal-conditioned policies that respects LTL semantics. (Leahy et al., 2024) recently proposed run-time task composition with safety semantics, which enables dynamic composition of pre-trained policies while maintaining safety guarantees. While their work focuses on safety properties, our approach extends compositional policy execution to general LTL specifications through a novel framework that decomposes formulas into sequences of sub-goals.

C ENVIRONMENT CONSTRUCTION

We provide additional detail on our formal environment definition and construction. We denote the agent’s environment $E \subseteq \mathbb{R}^n$. The environment contains non-intersecting regions $R \subseteq E$ taking labels from 2^Σ , where Σ is a set of atomic propositions corresponding to properties of interest in the environment (see Fig 3a). We define a labeling function $L : \mathcal{R} \rightarrow 2^\Sigma$, that defines which regions are labeled with which properties, where \mathcal{R} is the set of all labeled regions.

We further model the agent’s environment as a deterministic labeled Markov decision process (MDP), defined by the tuple $(\mathcal{S}, \mathcal{A}, \rho, r)$, where \mathcal{S} is the state space, \mathcal{A} is the action space, ρ is the Markov transition kernel $(s \in \mathcal{S}, a \in \mathcal{A}) \mapsto \rho(s, a)$ from $\mathcal{S} \times \mathcal{A}$ to \mathcal{S} , and r is the reward function. Since our pruned TS is deterministic, following our method in Section 3.2, it is consistent with our assumption of a deterministic labeled MDP.

An *execution* of a labeled MDP is the word τ . To transform an agent’s interaction with the environment into a set of labels that can be reasoned over for specification of ϕ , we *project* an execution onto the set of associated labels: $\downarrow_L: \mathcal{S} \times 2^\Sigma \rightarrow 2^\Sigma$. This means for a word $\tau = \langle s_0, l_0 \rangle, \langle s_1, l_1 \rangle, \dots, \langle s_i, l_i \rangle$ the projection is $\downarrow_L = l_0, l_1, \dots, l_i$ (Leahy et al., 2024). The sequence of non-empty symbols, the only ones we use for satisfaction purposes, is denoted $\downarrow_L^+(\tau)$. For example, if $\downarrow_L(\tau) = \emptyset, \emptyset, a, b, \emptyset, (a, c)$, then $\downarrow_L^+(\tau) = a, b, (a, c)$.

Definition 3 *LT Specification Satisfaction for Execution: For an LTL specification ϕ over Σ and a word τ , $\tau \models \phi$ iff $\downarrow_L^+(\tau)[-1] \models \phi$ (Leahy et al., 2024).*

For example, if the trained task is to produce the symbol c , the task is only satisfied if the last symbol in τ is c .

The labeling function of an MDP includes a set of goals: $\mathcal{G} \subseteq \mathcal{S}$. For a set of tasks \mathcal{M} , each task $\mu \in \mathcal{M}$ shares the same state space, action space, and environment transition dynamics (Nangue Tasse et al., 2020). The only difference between the tasks is the reward function, which is based on the sequence of symbols encountered along with the absorbing set $\mathcal{G} \subseteq \mathcal{S}$.

D BOOLEAN COMPOSITION AND MINIMUM VIOLATION SEMANTICS

D.1 BOOLEAN COMPOSITION

We employ Boolean task algebra in order to perform task conjunction \wedge and disjunction \vee (Nangue Tasse et al., 2020). Boolean task algebra allows the agent to use a pretrained set of primitive tasks to expand the number of tasks that can be achieved with no additional learning by expressing the additional tasks as a Boolean expression over the original task primitives.

We provide a brief introduction to Boolean task algebra. A set of optimal Q-value functions (and approximations) can be constructed as a Boolean algebra over the set of tasks \mathcal{M} (Nangue Tasse et al., 2020). \bar{Q}^* is the set of optimal extended \bar{Q} -value functions for $\mu \in \mathcal{M}$. Considering two distinct tasks, μ_0 and μ_1 , operators \neg , \vee , and \wedge are defined as the following.

Definition 4 (Nangue Tasse et al., 2020) *Negation, $\neg: \bar{Q}^* \rightarrow \bar{Q}^*$*

$\bar{Q}^* \mapsto \neg \bar{Q}^*$, where $\neg \bar{Q}^*: \mathcal{S} \times \mathcal{G} \times \mathcal{A} \rightarrow \mathbb{R}$

$$(s, g, a) \mapsto (\bar{Q}_1^*(s, g, a) + \bar{Q}_2^*(s, g, a)) - \bar{Q}^*(s, g, a)$$

Definition 5 (Nangue Tasse et al., 2020) *Conjunction, $\wedge: \bar{Q}^* \times \bar{Q}^*$*

$(\bar{Q}_1^*, \bar{Q}_2^*) \mapsto \bar{Q}_1^* \wedge \bar{Q}_2^*$, where $\bar{Q}_1^* \wedge \bar{Q}_2^*: \mathcal{S} \times \mathcal{G} \times \mathcal{A} \rightarrow \mathbb{R}$

$$(s, g, a) \mapsto \min \{ \bar{Q}_1^*(s, g, a), \bar{Q}_2^*(s, g, a) \}$$

Definition 6 (Nangue Tasse et al., 2020) *Disjunction, $\vee: \bar{Q}^* \times \bar{Q}^*$*

$(\bar{Q}_1^*, \bar{Q}_2^*) \mapsto \bar{Q}_1^* \vee \bar{Q}_2^*$, where $\bar{Q}_1^* \vee \bar{Q}_2^*: \mathcal{S} \times \mathcal{G} \times \mathcal{A} \rightarrow \mathbb{R}$

$$(s, g, a) \mapsto \max \{ \bar{Q}_1^*(s, g, a), \bar{Q}_2^*(s, g, a) \}$$

D.2 MINIMUM VIOLATION

All policies are trained using MV semantics from Leahy et al. (2024). A brief summary of the penalty-enforced semantics we employed for both environments are as follows.

Previous work introduces a notion of proper policies that ensure certain behavior within the entire MDP execution, word τ (Leahy et al., 2024). The safe path definitions are inspired by temporal logic (TL) planning, where certain behavior is specified for the entire $\downarrow_L^+(\tau)$ (Leahy et al., 2024).

Proper policies are policies that, given a specification ϕ , are guaranteed to reach a satisfying state (Leahy et al., 2024). This means the word τ created through the MDP policy execution satisfies ϕ : $\tau \models \phi$. The policies should avoid traveling through unsafe states.

Definition 7 *Unsafe State*: any state that is undesired, where a state $s \in \mathcal{S}$ such that $L(s) \not\models \phi$.

Formally, we define two types of proper policies in respect to LTL specification ϕ over Σ as follows (Leahy et al., 2024):

Definition 8 *Pure Path*: An execution τ generates a pure path if $|\uparrow_L^+(\tau)| = 1$ and $\tau \models \phi$. Intuitively, a pure path is a proper path that does not produce any symbols except the symbols that satisfy ϕ at the end of the execution. (Leahy et al., 2024)

Definition 9 *Minimum Violation (MV) Path*: An execution τ generates a MV path if $|\uparrow_L^+(\tau)| > 1$ and $\tau \models \phi$, and there is no execution τ' such that $|\uparrow_L^+(\tau')| < |\uparrow_L^+(\tau)|$. Intuitively, this means minimum violation paths produce as few non satisfying symbols as possible. (Leahy et al., 2024)

In order to enforce this behavior, any label generated that does not satisfy the current task is given a penalty. To enforce the multiple levels of behaviors to be avoided, the rewards are structured hierarchically, with less bad rewards for passing through an unsafe state than terminating in an unsafe state (Leahy et al., 2024). The full reward system is shown in Table 5, with R_{step} as a small negative value and R_{goal} as a proportionally larger positive value. η is a reward multiplier. Since we are using deterministic labeled MDPs, η is the number of steps an agent may detour in order to avoid undesirable regions (Leahy et al., 2024). Table 5 also includes the reward function for a given task $R_p(s_i, a, g, s_{i+1})$, with $g \in \mathcal{G}$, $\sigma \in \Sigma$, and done representing the end of an episode due to the agent terminating its execution.

| Symbol | Penalty Type | Reward Function | Value |
|-----------------|-------------------|--|-------------------|
| $R_{worstterm}$ | worst termination | $s_{i+1} \neq g \wedge done$ | $\eta^3 R_{step}$ |
| $R_{badterm}$ | bad termination | $s_{i+1} = g \wedge \sigma \notin l_i \wedge done$ | $\eta^2 R_{step}$ |
| $R_{badstep}$ | bad pass through | $s_{i+1} \neq g \wedge \neg done$ | ηR_{step} |
| R_{goal} | goal | $s_{i+1} = g \wedge \sigma \in l_i \wedge done$ | R_{goal} |
| R_{step} | step | otherwise | R_{step} |

Table 5: Reward hierarchy and corresponding reward function (Leahy et al., 2024).

Remark 2 For the office world environment, terminating in a state means selecting the action stay in a state. For the video game environment, it means having the agent "collect" the object.

E TRANSITION SYSTEM CONSTRUCTION AND PRUNING

E.1 GENERATING THE TRANSITION SYSTEM

As described in Section 2, adjacent regions $r_i, r_j \in \mathcal{R}$ are joined by an edge if they touch each other. However, each edge still requires a label defining the policies that will cause the agent to take that edge.

Algorithm 1 generates the transition labels for the TS. In this and the following algorithms, S is the set of states, $s \subseteq S$ is a state, T is the set of transitions, and $t \subseteq T$ is a transition. For each transition, the algorithm checks the distance (represented as function $d(\cdot)$) between the start and end state of that transition to each other state in the TS (lines 3–6). If the distance from the start state to a state is greater than the distance from the end state to a state, the transition is approaching that state (and its associated label), so that state’s label is added to the transition (lines 7–8).

A TS constructed in this manner could be non-deterministic, so we do not assume that the initial TS is deterministic. A *deterministic* transition system is a TS in which the transition relation $\rightarrow \subseteq S \times Act \times S$ is deterministic.

Definition 10 A TS is a deterministic transition system (DTS) if, given s_i and α_i , for every $(s_i, \alpha_i, s_{i+1}) \in \rightarrow$, s_{i+1} is unique.

Algorithm 1 Generate Transition System Transition Labels

```

1: procedure GENERATETSLABELS( $S, T$ )
2:   initialize transition labels  $L$ 
3:   for  $t \in T$  do  $\triangleright t = (startState, endState)$ 
4:     for  $s \in S$  do
5:        $d_{start} = d(startState, s)$ 
6:        $d_{end} = d(endState, s)$ 
7:       if  $d_{start} > d_{end}$  then
8:          $L[t] \leftarrow L[t] \cup L[s]$ 
9:   return  $L$   $\triangleright$  The transition system labels

```

An example of non-determinism in our TS is state q_2 in Fig. 3b has multiple outgoing transitions labeled a . Given $s_i = q_2$ and $\alpha_i = a$, s_{i+1} is not unique, as s_{i+1} could be either q_1 or q_3 . Our TS must not only accurately reflect the results of applying a given task policy from each state, but also be deterministic.

E.2 TRANSITION SYSTEM PRUNING

To to address nondeterminism, we introduce a TS pruning method, which removes symbols from transition labels. Algorithm 2 shows the overall pruning procedure. Each `case` function in the algorithm is explained in Section 3.2, along with an explanation of the results of pruning the TS from Figure 3b.

Algorithm 2 Transition System Prune

```

1: procedure PRUNE( $S, T, \Sigma$ )
2:    $S, T \leftarrow \text{case1}(S, T, \Sigma)$ 
3:   for  $s \in S$  do
4:      $T \leftarrow \text{case2}(s, T, \Sigma)$ 
5:      $T \leftarrow \text{case3}(s, T, \Sigma)$ 
6:    $T \leftarrow \text{emptyCleanup}(T)$ 
7:   return  $S, T$ 

```

E.2.1 CASE 1

Recall Case 1 from 3.2, which removes equivalent states from the TS:

Case 1: If a bisimulation TS_{\sim} exists for TS , reduce the total number of states by using TS_{\sim} .

The TS may contain multiple branches from a parent state which contain the same state and transition labels. In order to simplify the TS, we implement finite abstractions, which allow us to reduce the size of our finite TS while keeping details required for analysis and control. One such abstraction is Bisimulation, which uses observational equivalence to reduce the TS (Belta et al., 2017).

Definition 11 States $s_1, s_2 \in S$ are observationally equivalent ($s_1 \sim s_2$) iff $L(s_1) = L(s_2)$. (Belta et al., 2017)

Using the observational equivalence relation we can construct a quotient TS.

Definition 12 A quotient transition system (TS_{\sim}) is a tuple, $TS_{\sim} = (S_{\sim}, Act, \rightarrow_{\sim}, I, AP, L_{\sim})$, where

- S_{\sim} is a finite set of states which is the quotient space (set of equivalence classes);
- Act is a finite set of actions inherited from the original TS;
- $\rightarrow_{\sim} \subseteq S \times Act \times S$ is a transition relation. For $S_i, S_j \in S_{\sim}$, input $a \in Act$, and $s_1, s_2 \in S$, we include a transition $S_j \in \rightarrow_{\sim}(S_i, a)$ iff for states $s_1 \exists S_i$ and $s_2 \exists S_j \mid s_2 \in \rightarrow(s_1, a)$;

- $I \in S_\sim$ is an initial state;
- AP is a set of atomic propositions inherited from TS ; and
- $L_\sim : S_\sim \rightarrow 2^{AP}$ is a labeling function. For a $S_i \in S_\sim$ and $\forall s \in S$, the transition $L_\sim(S)$ is given by $L_\sim(S_i) = L(s)$

Given an equivalence class $S_i \in S_\sim$, the set of all equivalent states of TS are defined as $con(S_i) \subseteq S$, where each $con(S_i) \subseteq S$ denotes a set of all states with the same label. That is $\forall s \in con(S_i)$, $L(s) = L_\sim(S_i)$ (Belta et al., 2017).

The quotient TS_\sim can produce any word w_o that can be produced by the original TS; therefore, since any behavior of TS can be reproduced by TS_\sim , T_\sim simulates TS (Belta et al., 2017). This guarantees that if an LTL specification is satisfied for $S_i \in TS_\sim$ then the specification will be satisfied ($\forall s \in TS$) $\in con(S_i)$. Our \sim is a *bisimulation*, as every word produced by TS_\sim can be produced by TS .

Definition 13 A bisimulation equivalence relation \sim induced by our labeling function L is a bisimulation of a transition system $TS = (S, Act, \rightarrow, I, AP, L)$ if for all states $s_1, s_2 \in S$ and $\forall a \in Act$, if $s_1 \sim s_2$ and $s'_1 \in \rightarrow(s_1, a)$ then there exists $s'_2 \in \rightarrow(s_2, a)$ and $s'_1 \sim s'_2$

If \sim is a bisimulation, then TS_\sim is a actually a bisimulation quotient of TS , denoted TS_\approx (Belta et al., 2017). The quotient we propose is a bisimulation quotient, hence we use TS_\approx ."

E.2.2 CASE 2

Recall Case 2 from 3.2, which removes ambiguous transitions from the TS:

Case 2: If any outgoing transitions from a state share a symbol in the transition label, only keep the symbol in the transition with the least distance to the state labeled with the shared symbol, according to MV semantics. If all the distances to the state that is labeled with the shared symbol are the same, remove the symbol from all the transition labels of the state.

Algorithm 3 shows the procedure for case2. For a given state, the algorithm finds the set of outgoing transitions labeled by a given symbol σ (lines 2–3). If there is more than one such transition, determines the state(s) with the largest distance according to MV semantics (lines 5–7) and removes σ from the corresponding transitions. The process repeats until there is at most one transition with label σ .

Algorithm 3 Case 2 Prune

```

1: procedure CASE2( $s, T, \Sigma$ )
2:   for  $\sigma \in \Sigma$  do
3:      $t_\sigma \leftarrow \{t \in T \mid \sigma \in L(t)\}$ 
4:     while  $|t_\sigma| \geq 2$  do
5:        $s_\sigma \leftarrow \{s \in S \mid \sigma \in L(s)\}$ 
6:        $d_t \leftarrow \max_{(t, s_\sigma)}(d(s, s_\sigma))$ 
7:        $t_{far} \leftarrow \{t \in t_\sigma \mid d_t \text{ is greatest}\}$ 
8:       delete  $\sigma$  from  $t_{far}$ 
9:       delete  $t_{far}$  from  $t_\sigma$ 
10:  return  $T$ 

```

E.2.3 EMPTY CLEANUP

We require final cleanup after TS_\approx has been generated through the 3 cases. Since there will never be a policy for spaces that produce no symbols, the policy \emptyset is removed from all transition labels. Also, all transitions with no transition label are removed, as there is no policy that can take the agent between the connected states. The transition will create uncertainty when the product is taken with TS_\approx , so it is imperative it is removed. Figure 4d highlights the changes in TS_\approx after emptyCleanup in Alg. 2 is executed.

F PRODUCT BETWEEN TRANSITION SYSTEM AND BÜCHI AUTOMATON

Given a fully pruned TS_{\approx} with labels from Σ , we create a Büchi automaton using an LTL specification ϕ over Σ .

Definition 14 *Büchi Automaton:* A nondeterministic Büchi automaton is $B = (X, X_0, AP, \rightarrow, F)$, where

- X is a finite set of states;
- $X_0 \subseteq X$ is the set of initial states;
- AP is the input alphabet, a set of atomic propositions;
- $\rightarrow \subseteq X \times AP \times X$ is a nondeterministic transition function;
- $F \subseteq X$ is the set of final accepting states. (Belta et al., 2017)

A Büchi automaton is deterministic if X_0 is a singleton and $\rightarrow(x, \alpha)$ is either \emptyset or a singleton for all $x \in X$ and $\rho \in AP$ (Belta et al., 2017).

A product automaton (PA) allows us to combine the TS_{\approx} encoding of the environment and policies with the Büchi automation encoding of the LTL specification ϕ ; therefore, the PA captures the relationship between the behavior our RL policies can produce and the requirements defined by ϕ . Since we have an input to the TS_{\approx} , $I \in S_{\sim}$, our TS_{\approx} is a controlled transition system, so our resulting PA between the TS and Büchi automation is controlled (Belta et al., 2017). Since both our pruned TS_{\approx} and Büchi automation are deterministic, our resulting PA is also deterministic.

Definition 15 *Product Automaton:* The controlled Büchi product automaton $PA = TS \otimes B$ of a finite controlled transition system $TS = (S, Act, \rightarrow, I, AP, L)$ and a Büchi automaton $B = (X, X_0, AP, \rightarrow_B, F)$ is $PA = (X_P, X_{0P}, Act, \rightarrow_P, F_P)$ where

- $X_P = S \times X$ is the set of states;
- $X_{0P} = I \times X_0$ is the set of initial states;
- Act is a finite set of controls or actions;
- \rightarrow_P is the transition function where, for a state $(s, x) \in X_P$, we have $\rightarrow_P((s, x)) = \{(s', x') \in X_P \mid s' \in \rightarrow(s) \text{ and } x' \in \rightarrow_B(x, L(s))\}$;
- $F_P = S \times F$ is the set of accepting states.

(Belta et al., 2017)

Remark 3 Definition 15 for the PA differs slightly from the standard definition of an uncontrolled Büchi automaton definitions. Since there is no initial state for the TS for uncontrolled PAs, the uncontrolled definition $X_{0P} = S \times X_0$ (Belta et al., 2017) becomes $X_{0P} = I \times X_0$.

Given a controlled Büchi PA, $PA = (X_P, X_{0P}, Act, \rightarrow_P, F_P)$, one can find the largest set of initial states $W_{0P} \subseteq X_{0P}$ for which there exists a control function $\pi_P : X_P \rightarrow AP$ such that each run of PA under strategy (W_{0P}, Π_P) satisfies the Büchi accepting condition F_P (Belta et al., 2017).

For the controlled Büchi automaton PA generation and the control function search, we use LOMAP, <https://github.com/wasserfeder/lomap>. Each $\pi_P \in \Pi_P$ generates a symbol $\rho \in AP$. Per Definition 3, the process of executing each $\pi_P \in \Pi_P$ in sequence creates a satisfying word, where the agent’s behavior in the environment satisfies the LTL specification.

Product Example We show a complete example of the TS to PA to execution pipeline, to provide insight into our approach. Figure 10a shows the TS created from start state 2 of the video game environment, shown in Figure 13b. Figure 10a is the same as Figure 8a, but we repeat it to show each step of the PA generation process.

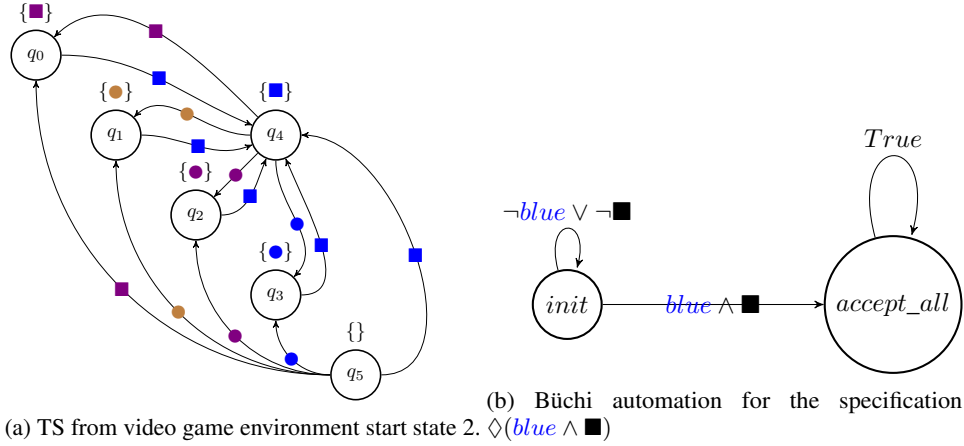
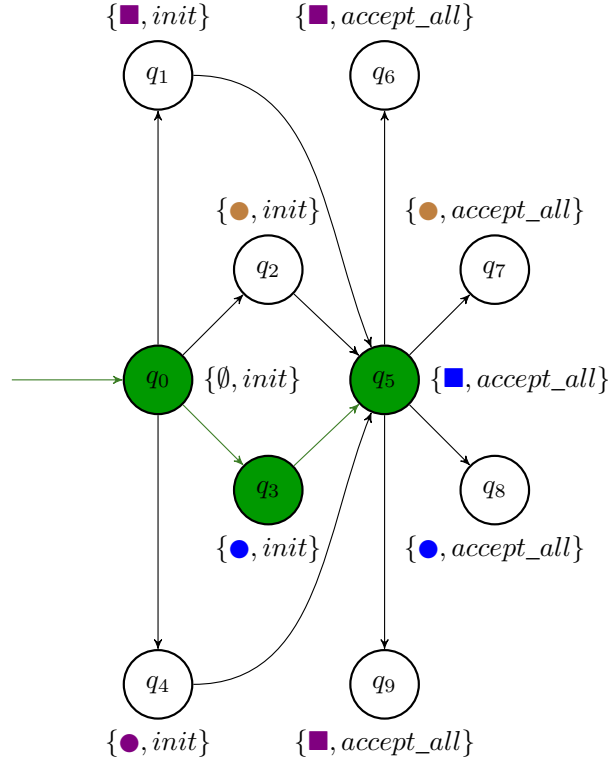


Figure 10

Figure 11: PA from TS of start state 2 and Büchi of specification $\Diamond(blue \wedge \blacksquare)$. The shortest word through the PA, τ , is denoted in green.

The corresponding PA for the product between the TS and Büchi automaton is shown in Figure 11.

The trace for shortest word τ for the PA for TS of start state 2 and Büchi of specification $\Diamond(blue \wedge \blacksquare)$ is shown in Figure 11. The shortest word is the shortest path from an $init$ state to an $accept_all$ state. The policies the agent executes in sequence are every $\rho \in AP$ that is in the PA node label. Because the agent's initial region is known, the initial state of the PA is also known. In this case, for start state 2, the agent begins in \emptyset ; therefore, $\uparrow_L^+(\tau) = (blue, \bullet), (blue, \blacksquare)$. To combine the $\rho \in AP$ policies, we use composition per (Leahy et al., 2024). Therefore, our policy list to execute, in sequential order, to satisfy the LTL specification $\Diamond(blue \wedge \blacksquare)$, is $[blue \wedge \bullet, blue \wedge \blacksquare]$.

G THEORETICAL ANALYSIS

In this section, we provide proof sketches for three theorems about our method.

Recall Theorem 1:

Theorem 1 *The resulting pruned TS from Sec. 3.2 is deterministic.*

Proof (sketch) 1 *We note two ways in which a TS may be non-deterministic. First, a TS may transition to multiple states given a single action (either indistinguishable from each other or not), which cases 1 and 2 address. Second, given a state and an action, the TS may stay in the same state or transition to another state. The MV semantics of our policies preclude moving in favor of self-loops, which case 3 addresses.*

Recall Theorem 2:

Theorem 2 *The resulting pruned TS from Sec. 3.2 contains no unrealizable transitions.*

Proof (sketch) 2 *The naive TS construction captures all potential transitions that an RL policy enforces. Our pruning process respects the MV semantics as defined in 2. By pruning transitions to farther states with the same label, we prune states that are unreachable under MV semantics, because those semantics prioritize producing fewer symbols.*

Recall Theorem 3:

Theorem 3 *Satisfying an LTL specification using product construction with our pruned TS_{\approx} is sound, meaning there are no false positives.*

Proof (sketch) 3 *This follows directly from Theorems 1 and 2. Those theorems imply that the agent executing its RL policies is a simulation of the pruned TS, and the usual guarantees on the PA hold; therefore finding a satisfying τ in the PA using our TS_{\approx} is sound.*

H RESULTS

H.1 COMPARISON ENVIRONMENT SETUP

Office World Environment Since our map does not change during training, and the observations are the agent’s location within the map, we do not change the map for testing. The map is shown in Figure 12.

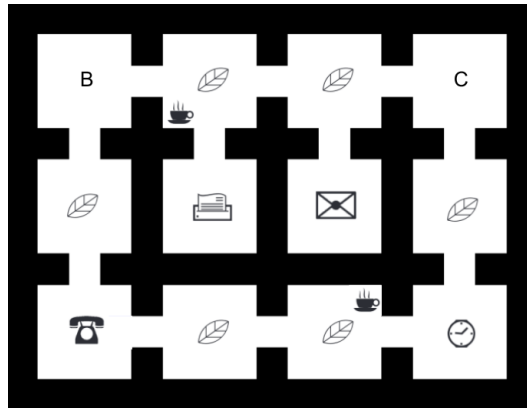


Figure 12: Office world map.

Video Game Environment When running comparisons we used one of two start states. The first, shown in Figure 13a, contains a symbol free corridor along the top of the map, while the second, shown in Figure 13b, does not. We use one of these two environments when generating results, to test our pure path and MV logic.

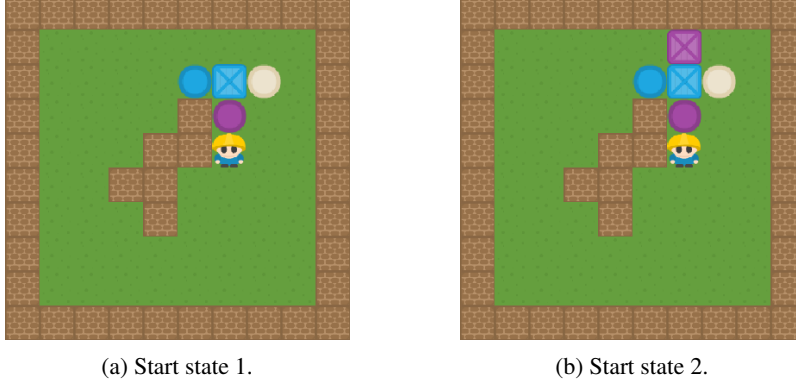


Figure 13: Two different map configurations used for the video game testing environment.

Continuous Environment When running comparisons we used one environment configuration with 6 different objects dispersed, shown in Figure 14

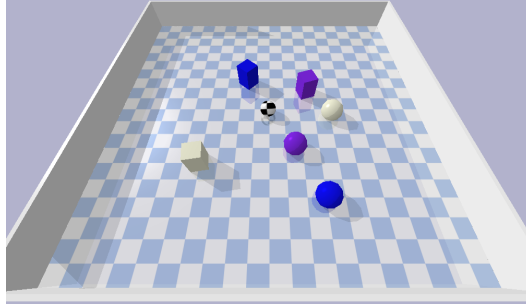


Figure 14: Continuous environment configuration.

H.2 SOFTWARE

Our simulations are executed in Python 3.7.

H.3 TRAINING THE POLICIES

Office World and Video Game Environments Both the office world environment and video game environment use the same reward hierarchy and values. The actual numerical values returned for the reward used during each training step are shown in Table 6, with $r_{min} = -0.1$ and $r_{max} = 2.0$.

The office world environment is trained on a laptop running Ubuntu 20.04, 12 cores CPU, 16 gb RAM, and GeForce GTX 1070 Mobile GPU. The video game environment is trained on a desktop running Ubuntu 22.04 with 32 cores CPU, 64 gb RAM, and A5500 GPU.

Continuous Environment The actual numerical values returned for the reward used during each training step are shown in Table 7, with $r_{min} = -0.005$, $r_{max} = 2.0$, and $n = \sqrt[3]{2 \div |r_{min}|}$

The continuous environment, like the video game environment, is trained on a desktop running Ubuntu 22.04 with 32 cores CPU, 64 gb RAM, and A5500 GPU.

| Symbol | Numeric Value |
|-----------------|---------------------|
| $R_{worstterm}$ | $20 \times r_{min}$ |
| $R_{badterm}$ | $15 \times r_{min}$ |
| $R_{badstep}$ | $10 \times r_{min}$ |
| R_{goal} | r_{max} |
| R_{step} | r_{min} |

Table 6: Actual values used during training for penalty hierarchy.


| Symbol | Numeric Value |
|-----------------|----------------------|
| $R_{worstterm}$ | $n^3 \times r_{min}$ |
| $R_{badterm}$ | $n^2 \times r_{min}$ |
| $R_{badstep}$ | $n \times r_{min}$ |
| R_{goal} | r_{max} |
| R_{step} | r_{min} |

Table 7: Actual values used during training for penalty hierarchy.

H.3.1 COMP-LTL MV POLICIES

Tabular Q-learning in Office World We train our tabular Q-learning MV policies in the office world environment. Office world is a 2D static grid world. Each grid cell is either unoccupied, represented by empty set \emptyset , or contains an one or more symbols associated with an item and quadrant of the board. One policy is trained for each of the eight characters plus each of the four quadrants of the grid.

During training the environment randomly spawns the player’s start position. The observation space is the agent’s location in the world (row, column). The action space is up, down, left, right, and stay. The optimal policies are calculated using value iteration.

We extended our environment to include training on all four quadrants of the environment as goals. Therefore, $\{\uparrow, \downarrow, \leftarrow, \rightarrow\}$ are added to the set of goals G . For example, if there are two  in the environment, the user may prefer the one on the top of the office environment. Therefore, they could specify $\square(\text{robot} \wedge \uparrow)$. By leveraging zero-shot compositionality this way, goals can be as specific as necessary to meet a user’s needs.

Safety penalties are used throughout the entire training process.

The hyperparameters used for training are:

- gamma: 1
- epsilon: 1
- alpha: 1

Deep Q-Learning in Video Game Environment We train our Deep Q-Learning (DQN) MV policies in the video game environment. The video game environment is a 2D item collection grid world where the items change location. Each grid cell is either unoccupied, represented by empty set \emptyset , or contains an object characterized by the conjunction of a shape and a color from the set of propositions $\{\text{w (white)}, \text{b (blue)}, \text{p (purple)}, \bullet \text{ (circle)}, \blacksquare \text{ (square)}\}$.

During training the environment randomly spawns items and the player’s start position. The observation space is down sampled 84x84 RGB images of the world and the action space is the four cardinal directions: up, down, left right. One policy is trained for each of the five propositions above, which we denote as primitive tasks. Each of the five primitive task policies used in this experiment are trained using the Deep Q-Learning (DQL) RL methodology from Leahy et al. (2024) with MV safety semantics to approximate the optimal policy.

We used a curriculum training approach for every primitive task model. We train each model to reach uniformly sampled goal items without any penalties (or use a pre-trained model from (Nangue Tasse et al., 2020)). When training we allocate 2M steps but the models converge by around 1M steps.

Next, we refine the models further by training on random environments with safety penalties. For this we also we allocate 2M steps and the models converge by around 1M steps again. Finally, we refine the model on closely spaced items, as the models perform well on random environments which tend to have spread out items, but they require additional training to handle environments with crowded objects. This fine tuning converges in 20-60k steps.

The hyperparameters used for training are:

- target update frequency: 1,000 steps
- learning rate: 1e-5
- batch size: 256
- replay buffer: 500,000
- gamma: 0.99

Twin Delayed DDPG in Continuous Environment We train our Twin Delayed DDPG (TD3) MV policies in the continuous environment. TD3 is an actor-critic network with mechanisms that minimize function approximation errors in both the actor and critic (Fujimoto et al., 2018). The continuous environment is an open arena with 3D objects placed throughout from the set of propositions {w (white), b (blue), p (purple), ● (sphere), ■ (box)}.

During training the environment randomly spawns objects. The observation space is 96D LIDAR-inspired observations and a continuous 2D force vector action space. The agent is represented as a ball with free movement along the force vector. 6 policies (one for each of the combinations of color and shape) are trained for each of the five propositions above, which we denote as primitive tasks. Each set of tasks for each primitive is trained using TD3 with MV safety semantics from Leahy et al. (2024) to approximate the optimal policies.

This approach does not use curriculum training and uses safety penalties throughout the training. 4M training steps are allocated to training on randomized object locations, with an additional 1M fine tuning step on a static demo environment.

The hyperparameters used for training are:

- target network update rate: 5e-3
- evaluation frequency: 5e3
- actor and critic batch sizes: 256
- gaussian exploration noise: 0.1
- gamma: 0.99
- policy noise: 0.2
- noise clip: 0.5
- policy frequency: 2

Policy noise is the noise added to the target policy during the critic update step and noise clip is the range to clip the target policy noise. The policy frequency is the frequency of delayed policy updates.

H.3.2 REWARD MACHINES

We train RM with q-learning and hierarchical reward machine (HRM) for both the office world environment video game environment.

For training the RMs for comparison, we use the default training parameters. To increase training efficiency and reduce sparse rewards, RMs has the option to use Counterfactual Experiences for Reward Machines (CRM) and Automated Reward Shaping (RS). CRM uses counterfactual reasoning to create synthetic experiences for the agent to learn policies faster (Icarte et al., 2018). RS changes the reward generated from a simple RM so the optimal policies are consistent but the overall reward is less sparse (Icarte et al., 2018). RS uses two hyperparameters when calculating the shaped rewards: 1) the environment discount factor, gamma, and 2) the discount factor used in the potential function, RS gamma (Icarte et al., 2018).

Q-learning The hyperparameters and settings used for Q-learning are as follows:

- number of timesteps: 1e5 steps
- gamma: 0.9
- RS gamma: 0.9
- use CRM: true
- use RS: true

HRM The hyperparameters and settings used for HRM are as follows:

- number of timesteps: 1e5 steps
- gamma: 0.9
- RS gamma: 0.9
- use CRM: false
- use RS: true

H.3.3 SKILL MACHINES

We train SM with tabular Q-learning for the office world environment and a deep Q network (DQN) for the video game environment. We train the primitives prior to running zero-shot.

For the few-shot approach, we train for 400,000 timesteps. Since we need to use few-shot in the office world environment, we few-shot using Q-learning

Tabular Q-learning The hyperparameters and settings used for Q-learning are as follows:

- number of timesteps: 1e5 steps
- gamma: 0.9
- learning rate (lr): 0.1
- epsilon: 0.5

DQN The hyperparameters and settings used for DQN are as follows:

- number of timesteps: 1e5 steps
- buffer size: 1e6
- exploration fraction: 0.5
- exploration final eps: 0.1