

Comp-LTL: Temporal Logic Planning via Zero-Shot Policy Composition

Taylor Bergeron¹, Zachary Serlin², and Kevin Leahy¹

¹Worcester Polytechnic Institute
100 Institute Rd
Worcester, MA 01609 USA
²MIT Lincoln Laboratory
244 Wood St
Lexington, MA 02421 USA

Abstract

This work develops a zero-shot mechanism, Comp-LTL, for an agent to satisfy a Linear Temporal Logic (LTL) specification given existing task primitives trained via reinforcement learning (RL). Autonomous robots often need to satisfy spatial and temporal goals that are unknown until run time. Prior work focuses on learning policies for executing a task specified using LTL, but they incorporate the specification into the learning process. Any change to the specification requires retraining the policy, either via fine-tuning or from scratch. We present a more flexible approach—to learn a set of composable task primitive policies that can be used to satisfy arbitrary LTL specifications without retraining or fine-tuning. Task primitives can be learned offline using RL and combined using Boolean composition at deployment. This work focuses on creating and pruning a transition system (TS) representation of the environment in order to solve for deterministic, non-ambiguous, and feasible solutions to LTL specifications given an environment and a set of task primitive policies. We show that our pruned TS is deterministic, contains no unrealizable transitions, and is sound. We verify our approach via simulation and compare it to other state-of-the-art approaches, showing that Comp-LTL is safer and more adaptable.

1 Introduction

A major goal in autonomous systems is the deployment of robots that are capable of executing complex tasks in safety-critical scenarios, such as search and rescue (SAR), transportation, and healthcare emergency services (Bogue 2019; Bravo and Leiras 2015). In these situations, the requirements may be time-varying, interdependent, and otherwise complex. For example, in healthcare, it is imperative to have a flexible “triage” approach to high priority tasks while still safely executing actions. Additionally, in a fire SAR scenario, certain rooms may become inaccessible, requiring a change of the environment representation and temporal task ordering for searching the building.

One approach to addressing such complex task executions is planning with linear temporal logic (LTL) (Baier and Katoen 2008; Kress-Gazit, Lahijanian, and Raman 2018). LTL allows a user to specify tasks with complex temporal and inter-task relationships. A major strength of this approach is the focus on correct-by-construction algorithms that are capable of planning for an arbitrary formula specified by

a user. However, many associated planning approaches require reliable task models in order to guarantee satisfaction of an LTL specification (Kress-Gazit, Lahijanian, and Raman 2018; Belta, Yordanov, and Gol 2017).

Recent work has focused on using reinforcement learning (RL) to meet LTL specifications when a model of the system is unavailable. Many of these approaches incorporate an automaton or similar structure in the training process. One such approach is Reward Machines (Icarte et al. 2018) (RM), which encodes the specification as reward functions while exposing the reward function structure to the training agent. Other approaches use an automaton to guide the learning process in the presence of partially-satisfiable LTL specifications (Cai et al. 2023), or to learn sub-policies corresponding to edges in an automaton (Li et al. 2019). While all of these approaches learn policies that are capable of executing a high-level task specified using LTL, they incorporate the specification into the learning process, so any change to the specification requires retraining the policy. We desire a more flexible approach – to learn a set of policies that can be used to satisfy arbitrary specifications without retraining.

A closely related approach is Skill Machines (Tasse et al. 2024), which leverages prior work on zero-shot composition (Tasse, James, and Rosman 2020), to satisfy a proposition on a reward machine. While changing the specification does not require re-training from scratch, it nonetheless requires fine-tuning of the policies to guarantee satisfaction. In our work, we aim to find a solution that requires no retraining and no fine-tuning beyond the initial training of tasks, while still being able to satisfy an arbitrary LTL specification. Another closely related approach, LTL-Transfer (Liu et al. 2024), is a zero-shot LTL solution that adheres to safety specifications encoded in the LTL formula. LTL-Transfer creates a Büchi automaton representation of the specification, and trains on the transitions in the automaton and transfers the logic to new LTL specifications. This constrains the possibilities of LTL specifications that can be satisfied to the set of automaton transitions that have already been explored during the training pass, whereas we desire a more broadly applicable solution.

In this work, we propose a framework for finding a satisfying solution for an environment and specification regardless of the exact environment, specification, or policies.

Inspired by Kloetzer and Belta (2008) and recent work in zero-shot Boolean Composition (Tasse, James, and Rosman 2020) (BC), we observe that compositional approaches allow us to satisfy Boolean constraints on automaton representations of LTL specifications. We leverage our prior work on safety-aware Boolean compositions of primitive policies to ensure the solution can be run zero-shot (Leahy, Mann, and Serlin 2024), and that the satisfying word can be achieved in the environment. Figure 1 shows our approach, Comp-LTL.

The specific contributions of this work are the following:

1. We develop a method for abstracting a geometric representation of an environment into a transition system (TS) with transition labels representing feasible Boolean combinations of tasks to transition between regions;
2. We resolve nondeterminism in the transitions enabled by the Boolean composition of primitive task policies; and
3. We demonstrate that this representation allows zero-shot satisfaction of LTL specifications at run time.

We support our theoretical results with case studies in simulation and comparison to other approaches.

2 Background and Problem Formulation

We consider an agent moving in a planar environment according to a high-level mission description. We denote the agent’s environment $E \subseteq \mathbb{R}^2$. 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. For $\sigma \in \Sigma$, we assume an agent can perform task σ if it is in a region r of the environment such that $\sigma \in L(r)$. Given an agent’s motion through a sequence of regions r_0, r_1, \dots , the agent produces a word $\tau = L(r_0), L(r_1) \dots$

The agent’s task is specified using linear temporal logic (LTL) (Baier and Katoen 2008). LTL includes Boolean operators, such as AND (\wedge), OR (\vee), and NOT (\neg), along with time based operators *eventually* (\diamond), *always* (\square), and *until* (\mathcal{U}). The formal syntax of LTL in Backus–Naur form is

$$\phi ::= \top | \sigma | \neg\sigma | \phi_1 \vee \phi_2 | \phi_1 \wedge \phi_2 | \phi_1 \mathcal{U} \phi_2 | \diamond\phi_1 | \square\phi_1, \quad (1)$$

where $\sigma \in \Sigma$ is an atomic proposition, and ϕ , ϕ_1 , and ϕ_2 are LTL formulas (Baier and Katoen 2008).

Due to space constraints, we do not describe the semantics of LTL here, but provide a brief intuition. A sequence τ satisfies a specification ϕ (written $\tau \models \phi$) if the sequence matches the properties specified by ϕ . For example, if $\phi = \diamond\sigma$ (“eventually σ ”), $\tau \models \phi$ if σ occurs at some point in τ . Similarly, $\square\sigma$ (“always σ ”) requires that σ appear at every point in τ . Interested readers are directed to Baier and Katoen (2008) for more details on the semantics of LTL. Importantly, off-the-shelf software, such as SPOT, can automatically translate LTL specifications into Büchi automata (Duret-Lutz et al. 2022). Furthermore, each transition on such automata can be described by a Boolean combination of atomic propositions.

We assume there is no transition model for the agent in its environment, and therefore we use reinforcement learning (RL) to learn a model for the agent to execute its tasks. RL is a branch of machine learning that maps states to actions in order to maximize a reward function (Sutton and Barto 2018). To facilitate satisfaction of temporal logic objectives, we will leverage our prior work (Leahy, Mann, and Serlin 2024) on safety-aware task composition to train policies for a given set of tasks. In that work, we proposed a method for learning policies that have “minimum-violation” (MV) safety semantics. For a word τ , let the number of positions in the word with non-empty symbols be denoted $|\tau|$ and the set of symbols in the last position of the word be denoted τ_f . Then, for a Boolean formula φ , we define minimum-violation semantics as follows.

Definition 1 *Minimum-violation Path:* A word τ is a minimum-violation path if $|\tau| > 1$ and $\tau_f \models \varphi$ and there is no word τ' such that $|\tau'| < |\tau|$ (Leahy, Mann, and Serlin 2024).

Intuitively, a minimum-violation is a path that: 1) terminates in a state that satisfied a Boolean formula; 2) if possible, visits no additional labeled states; and 3) if not possible, visits the fewest additional labeled states.

Problem 1 *Given a set of labels Σ , an environment E labeled from Σ , train policies for achieving tasks corresponding to $\sigma \in \Sigma$ such that the policies can be used to satisfy an LTL formula ϕ over Σ without additional training.*

3 Technical Approach

To solve Problem 1 we introduce a novel policy-aware environment abstraction as described below. Figure 2 shows an overview of the proposed solution algorithms. First, we create a TS (Belta, Yordanov, and Gol 2017) that captures both the topology of the environment as well as the policies for each primitive task to move the agent between regions. Constructing such a TS is conservative and can introduce ambiguity and non-determinism. To that end, we identify 3 cases for pruning the edge labels to remove non-determinism due to the reliance on task composition. The resulting TS can be used for planning to satisfy an LTL specification in the standard method (Belta, Yordanov, and Gol 2017), while accurately capturing the behavior created by the RL policies.

Generating The Transition System

To facilitate reasoning about satisfaction of an LTL specification, we abstract the environment as a transition system (TS). A TS describes the discrete behavior of a system via states and transitions and is formally defined as follows.

Definition 2 *A transition system (TS) is a tuple, $TS = (S, Act, \rightarrow, I, AP, L)$, where*

- S is a finite set of states;
- Act is a finite set of actions;
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation;
- $I \in S$ is an initial state;
- AP is a set of atomic propositions; and
- $L : S \rightarrow 2^{AP}$ is a labeling function.

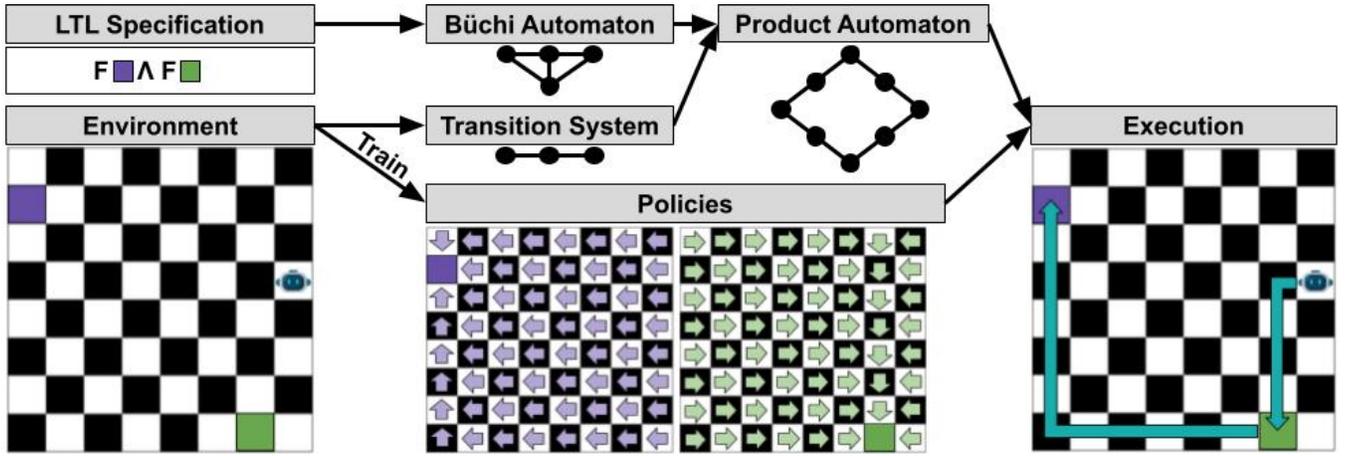


Figure 1: Comp-LTL training and path execution pipeline. Policies are trained for a set of primitive base tasks, and the primitive tasks are defined by the user based on knowledge of the tasks that will need to be executed in the environment. The environment is abstracted as a policy-aware transition system, which can then be composed with a Büchi automaton to plan over the Boolean composition of task policies.

We do not assume that the TS is deterministic. A *deterministic* transition system is a TS in which the transition relation $\rightarrow \subseteq S \times Act \times S$ is deterministic. For every $(s_i, \alpha_i, s_{i+1}) \in \rightarrow$, given s_i and α_{i+1} , s_{i+1} is unique.

To create the initial TS, each region is instantiated as a state, and adjacent regions are connected by transitions. This captures the topology of the environment. In planning- and control-based approaches, it is typical to assume that an agent can travel between any adjacent regions. For example, Fig. 3a shows an environment and a corresponding TS (3b). In a planning framework, an agent may choose which of the regions labeled a to visit. Using our RL approach, however, for an agent in the unlabeled region q_2 , executing a policy corresponding to a may cause the agent to visit q_1, q_3, q_5 , or q_6 , since the transition function is unknown. We introduce a pruning process to model and resolve such ambiguities.

First, our TS must accurately reflect the results of applying a given task policy from each state. Algorithm 1 generates the transition labels for the TS for this purpose. 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. For example, state q_2 in Fig. 4 has multiple outgoing transitions labeled a . To resolve this non-determinism, we propose a method for pruning the TS. To prune, we will remove transitions and policies in transition labels that introduce non-determinism, by checking for the specific cases of: 1) Equivalency; 2) Ambiguity; and 3) Feasibility; with the methods for mitigating these cases later described in Sec. 3.

Algorithm 1: Generate Transition System Transition Labels

```

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

```

Remark 1 We note that our approach differs from standard approaches to determinising a system, such as subset construction (Hopcroft, Motwani, and Ullman 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.

Transition System Pruning

When we follow the procedure outlined in Sec. 3, we capture how states are connected, but the resulting TS state and transition labels can introduce non-determinism. To mitigate such problems, 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 below, along with an explanation of the results of pruning the TS from Figure 4.

Case 1: Equivalent States In a TS where multiple branches from a parent state contain the same state and transition labels, taking a policy from one of the symbols shared on the transition label could take the agent to any of the duplicate child state regions. Since there is not meaningful distinction between the two for the purposes of satisfying a

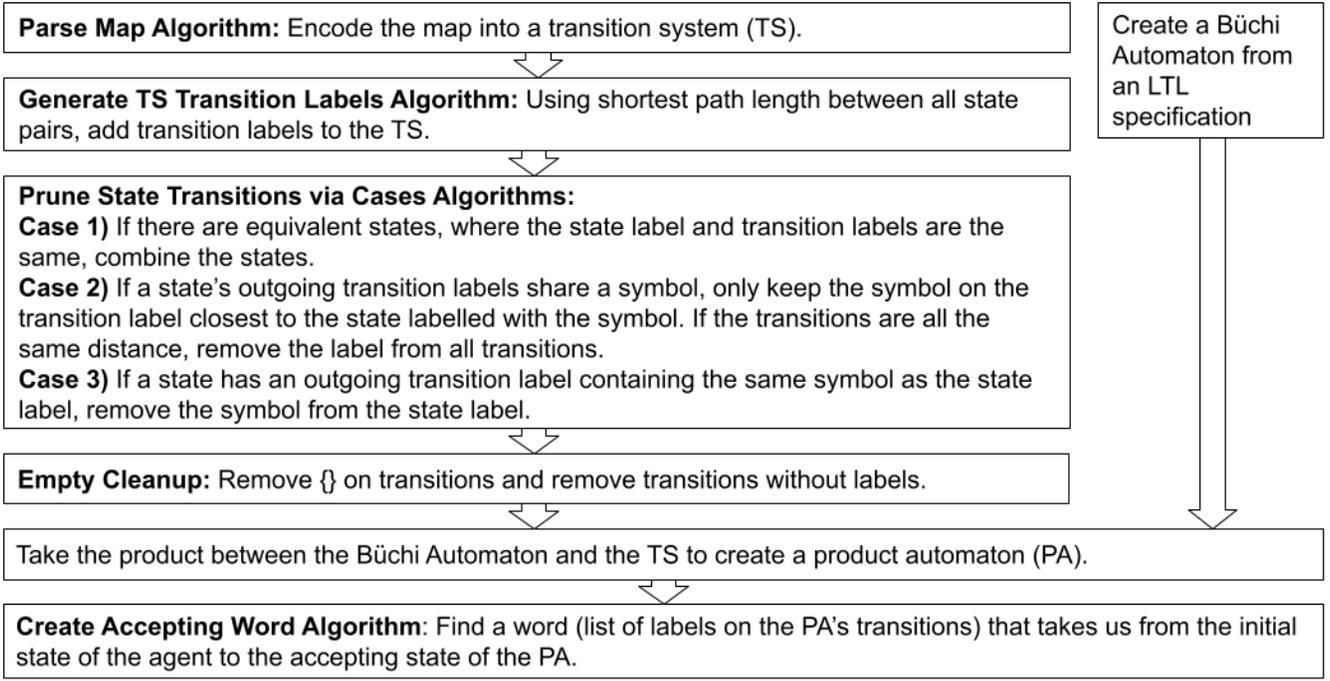


Figure 2: Block diagram representation of the overall Zero-Shot Algorithm for Comp-LTL. It includes the TS generation algorithm described in Section 3, from parsing the map, generating the initial TS, and pruning the TS to remove equivalency, solve ambiguity, and ensure feasibility. All sub algorithms are denoted in bold.

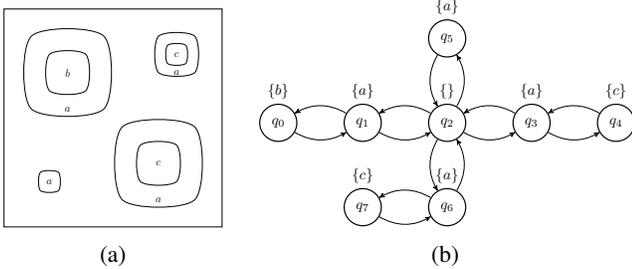


Figure 3: Example of an environment with regions to be parsed by Algorithm 1 (3a). A corresponding TS (3b) cannot be used directly, since it is not known a priori which region with a given label will be reached when executing a policy trained with RL.

specification, they are effectively equivalent. Therefore, all duplicate branches are merged into one to simplify the TS.

Case 1: If there are duplicate states, where the state label, outgoing transition labels, and incoming transition labels are the same between states, then combine the states into one.

Figure 5 highlights the changes in the TS after `case1` in Alg. 2 is executed. Algorithm `case1` identifies that q_2 has branches that are equivalent. The two equivalent branches are 1) the branch containing q_3 and q_4 and 2) the branch containing q_6 and q_7 . In Fig. 5, the two equivalent branches get combined into the branch containing q_3 and q_4 .

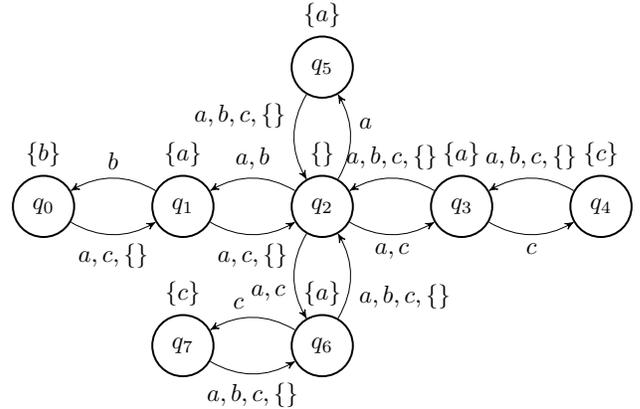


Figure 4: Unpruned TS created by Alg. 1 on the environment in Fig. 3a. State labels appear above each state. Transition labels correspond to task policies that enable a transition.

Case 2: Ambiguous Transitions If a state has multiple transition labels that contain the same symbol, it is uncertain which transition will be followed when the corresponding policy is executed. Because we seek a method that is zero-shot, we perform no additional checks or training on the policy to see how it would behave if run in the state region; therefore, we wish to keep at most once outgoing transition labeled with that symbol.

Case 2: If any outgoing transitions from a state share a

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  $S, T$ 

```

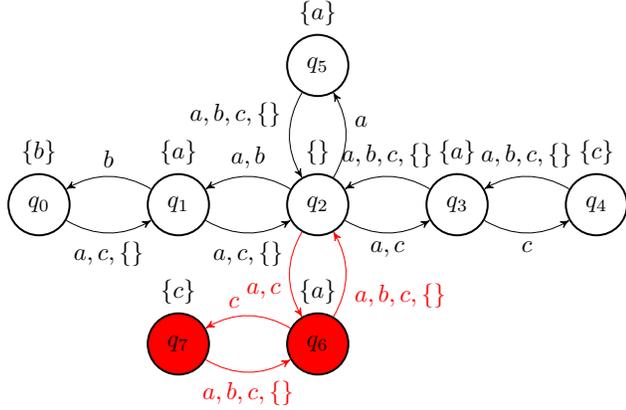


Figure 5: State labels appear above each state. Transition labels correspond to task policies that enable a transition. Red states and transitions are removed via `case1` to create Figure 6. Symbols a , b , c , $\{\}$ are deleted to remove equivalency in the system.

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  $T$ 

```

Figure 6 highlights the changes in the TS after `case2` is executed. No labels a are kept on outgoing transitions from q_2 , because MV semantics cannot distinguish them. The label c is removed from the transition linking q_3 to q_2 , because MV semantics will result in an agent transitioning from q_3 to q_4 under a policy associated with task c .

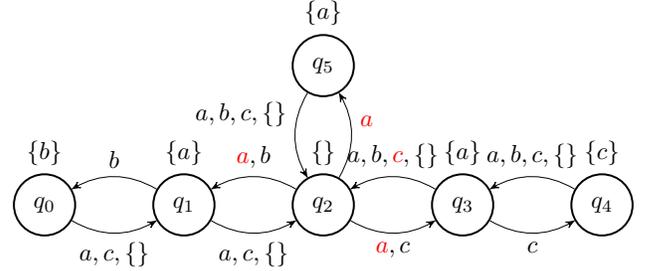


Figure 6: State labels appear above each state. Transition labels correspond to task policies that enable a transition. Red label symbols are removed via `case2` to create Figure 7. Symbols a and c are deleted to remove unambiguous transition labels from the system.

Case 3: Ineffectual Transitions and Feasibility This case only arises when there are multiple states containing the same symbol label during the initial TS creation (Alg. 1). Each duplicate state will have an outgoing transition label containing the same symbol as its own label, to get to the other states that share the same symbol label. We prune the symbol from the outgoing transition labels as running the policy for generating a symbol while already in the region that produces the symbol will not cause the agent to transition out of its current state. Therefore, since the state does not change, the symbol on the label is ineffectual.

Case 3: If a state shares the same label as any outgoing transition, remove the label from those transitions.

Figure 7 highlights the changes in the TS after `case3` in Alg. 2 is executed. State q_1 's label is a , and the transition from q_1 to q_2 contains a , so a is removed from that transition. The same logic applies to the other highlighted labels.

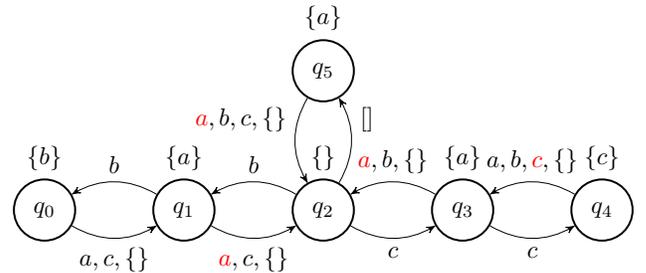


Figure 7: State labels appear above each state. Transition labels correspond to task policies that enable a transition. Red label symbols are removed via `case3` to create Fig. 8. Symbols a and c are deleted to remove system infeasibility from the system.

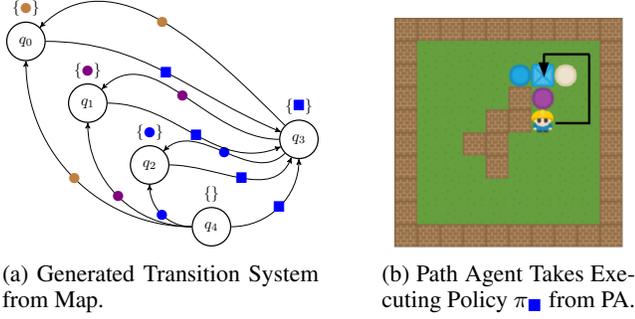


Figure 9: Pipeline for $\diamond \blacksquare$.

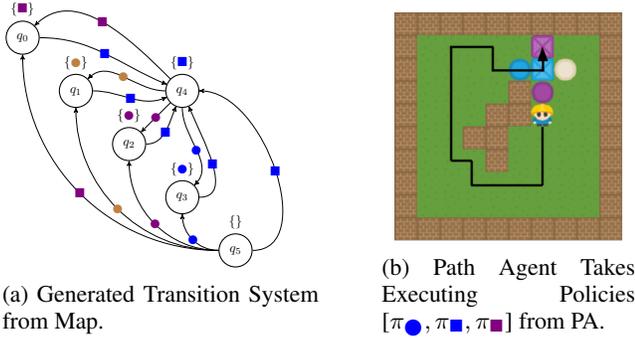


Figure 10: Pipeline for $\diamond(b \wedge \neg \blacksquare) \wedge \diamond p$.

The agent progresses along the list of policies in the order provided, so first the agent begins executing π_{\bullet} . Once the agent has reached a region that produces $b \wedge \bullet$, the agent transitions to executing the next policy. The agent is done when it has reached a region that produces the symbols of the final policy. In this instance, the agent is finished when it enters the region that produces $p \wedge \blacksquare$.

Since the agent is trained using MV semantics, Fig. 10b shows that the agent following a policy does not enter any cell containing another color or shape. That is why the first path to the blue circle takes the agent the long way around the center obstacle, as it will not travel the shorter path to blue circle and encounter additional symbols from the other region; therefore the path is optimal following our logic as there are not any additional symbols encountered along the path and the path never violates the LTL specification.

Remark 2 A trade-off of our approach is demonstrated in this case study. The agent does not take the shortest path in the environment, $\{\bullet, \blacksquare, \bullet\}$, since we only consider path length in the automaton. The paths $\{\bullet, \blacksquare, \bullet\}$ and $\{\bullet, \blacksquare, \blacksquare\}$ both have automaton path length 3. This is one of the primary trade-offs for zero-shot satisfaction, and methods such as RM can use fine-tuning to address this trade-off, but require additional training episodes.

Primitive Policy Training Time		
Policy Type	MV Time (s)	BC Time (s)
Blue	270,602	224,564
White	272,423	192,312
Purple	321,445	237,616

Table 1: Time to train primitive models for object colors.

Execution Time for a New LTL Specification		
LTL Specification	Comp-LTL Time (s)	RM (s)
$\diamond(b \& \blacksquare)$	0.0371	19151
$\diamond(p \& \bullet)$	0.0422	30647
$\diamond b \& \square \neg \blacksquare$	0.0552	1336
$\square(\diamond(b \& \blacksquare)) \& \square(\diamond(p \& \bullet))$	0.0305	3959

Table 2: Time to reprocess given a new LTL specification.

Comparison

Our main contribution is an approach that is safety aware and zero-shot. We compare our approach, Comp-LTL, to two other state-of-the-art approaches BC and RM. Comp-LTL trains tasks primitives using safety properties before run time and combines models temporally as needed using composition at run time (zero-shot) using environmental information to satisfy the specification. These safety-focused policies are MV policies. BC (Tasse, James, and Rosman 2020) trains task primitives before run time and combines models as needed using composition. RM (Icarte et al. 2018) trains a task policy at run time based on a specification provided at run time.

To demonstrate the necessity of safety primitive policies, we train primitive policies using BC and replace our safety primitive policies in our pipeline with their primitive policies. To demonstrate the run time benefits of zero-shot composition, we implemented RM on the same Boxman environment, trained using the default/recommended hyperparameters, and directly compare our entire pipeline’s performance to RM’s performance.

We compare the approaches based on three metrics 1) path safety; 2) training time; and 3) specification processing time. Path safety ensures that when a primitive policy, or composition of primitive policies, is being executed, no other symbol is produced unless necessary. Training time is the time for a primitive policy to be fully trained. This is not applicable for RM as there are no primitive policies to train. Specification processing time is the time for the approach to recalculate the approach based on a new LTL specification. We provide RM with the new LTL specification as a state machine which they train on, and we create the state machine from a translation of the specification automation created by SPOT (Duret-Lutz et al. 2022). The MV and BC approaches are provided a new LTL specification as a string.

Table 1 shows that MV primitive policies take longer to train than non-MV policies. Comp-LTL takes, on average, 32% longer than BC to train primitive policies.

Table 2 shows that upon a change in the LTL specification,

Number of “Unsafe” Symbols Collected			
LTL Specification	Comp-LTL	Comp-LTL + BC Policies	RM
$\diamond(b \& \blacksquare)$	0	1	1
$\diamond(p \& \bullet)$	0	0	0
$\diamond b \& \square \neg \blacksquare$	0	0	2
$\square(\diamond(b \& \blacksquare)) \& \square(\diamond(p \& \bullet))$	0	0	0
Total	0	1	3

Table 3: Number of symbols collected that are not specified in the policy or specification. Green indicates the specification was satisfied by the trajectory (agent behavior), and red indicates the specification was not satisfied by the trajectory. Comp-LTL with BC policies is our framework with our safety policies swapped for BC policies.

our mechanism takes significantly less time to reprocess, as we are not training whereas RM requires retraining. For a new LTL specification, our reward is, on average, 99.99% quicker than RM. We observe that reward machines that include a transition that uses \vee take longer to train than reward machines without \vee . The reward machines for specifications $\diamond(b \& \blacksquare)$ and $\diamond(p \& \bullet)$ include \vee , so they take on average 8.4% longer to train than the specifications $\diamond b \& \square \neg \blacksquare$ and $\square(\diamond(b \& \blacksquare)) \& \square(\diamond(p \& \bullet))$, which do not include \vee in their reward machines.

Table 3 shows that Comp-LTL’s additional training for safety results in no additional symbols being generated other than the symbol for the primitive policy and we are the only approach to consistently satisfy the specification. Table 3 also shows that other approaches produce extraneous symbols. For the specification $\diamond b \& \square \neg \blacksquare$, Comp-LTL with BC policies only collects the purple circle symbol, it does not reach any blue symbols, so it fails the specification even though it doesn’t produce any unsafe behavior. For the same specification RM collects purple circle and blue square, but square is explicitly not allowed, so it fails the specification. Finally, for the specification $\square(\diamond(b \& \blacksquare)) \& \square(\diamond(p \& \bullet))$, RM does not converge as the reward machine never collects any rewards – we note that we used the default recommended training parameters and with more parameter tuning or a different reward machine representation of the LTL specification the policy may have converged.

Our results show that our zero-shot approach requires no additional training per specification, and the paths our approach produces are safe and feasible.

5 Conclusion

We present Comp-LTL, an end-to-end zero-shot approach for executing an LTL task specification. To encode environment topology, we create a TS representation of the environment. Creating the TS introduces non-determinism and ambiguity, which we resolve via pruning. Our pruned TS is deterministic, contains only feasible transitions, and is sound. We create a Büchi automaton from an LTL specification then

take the product between the TS and Büchi automaton to create a PA, which encodes both environment topology and the task specification. We use the PA to find a satisfying path for the agent to reach an acceptance state in the LTL specification. This satisfying path is a list of MV policies for the agent to execute throughout its environment traversal. Comp-LTL is validated via simulation. We also show that our MV policies do not produce any extra symbols, unless necessary, and verify that the policies produce the expected symbols. We also compare processing and training time to other state of the art approaches, showing that our approach is safer and more adaptable.

Future work includes demonstrating the effectiveness of Comp-LTL on a variety of systems, including but not limited to a changing environment. When the agent receives feedback that the environment has changed, it can update the transition system to reflect the changes, and therefore update its solution to solving the LTL specification and the corresponding satisfying path. 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.

References

- Baier, C.; and Katoen, J.-P. 2008. *Principles of Model Checking*, volume 26202649. The MIT Press. ISBN 978-0-262-02649-9.
- Belta, C.; Yordanov, B.; and Gol, E. 2017. *Formal Methods for Discrete-Time Dynamical Systems*, volume 89. Springer. ISBN 978-3-319-50762-0.
- Bogue, R. 2019. Disaster relief, and search and rescue robots: the way forward. *Industrial Robot: the international journal of robotics research and application*, 46(2): 181–187.
- Bravo, R.; and Leiras, A. 2015. Literature review of the application of UAVs in humanitarian relief. *Proceedings of the XXXV Encontro Nacional de Engenharia de Producao, Fortaleza, Brazil*, 13–16.
- Cai, M.; Mann, M.; Serlin, Z.; Leahy, K.; and Vasile, C.-I. 2023. Learning Minimally-Violating Continuous Control for Infeasible Linear Temporal Logic Specifications. In *2023 American Control Conference (ACC)*, 1446–1452.
- Duret-Lutz, A.; Renault, E.; Colange, M.; Renkin, F.; Aisse, A. G.; Schlehuber-Caissier, P.; Medioni, T.; Martin, A.; Dubois, J.; Gillard, C.; and Lauko, H. 2022. From Spot 2.0 to Spot 2.10: What’s New? In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV’22)*, volume 13372 of *Lecture Notes in Computer Science*, 174–187. Springer.
- Hagberg, A.; Swart, P. J.; and Schult, D. A. 2008. Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States).
- Hopcroft, J. E.; Motwani, R.; and Ullman, J. D. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1): 60–65.

- Icarte, R. T.; Klassen, T.; Valenzano, R.; and McIlraith, S. 2018. Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning. In Dy, J.; and Krause, A., eds., *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, 2107–2116. PMLR.
- Kloetzer, M.; and Belta, C. 2008. A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Transactions on Automatic Control*, 53(1): 287–297.
- Kress-Gazit, H.; Lahijanian, M.; and Raman, V. 2018. Synthesis for Robots: Guarantees and Feedback for Robot Behavior. *Annual Review of Control, Robotics, and Autonomous Systems*, 1.
- Leahy, K.; Mann, M.; and Serlin, Z. 2024. Run-Time Task Composition with Safety Semantics. In *Forty-first International Conference on Machine Learning*.
- Li, X.; Serlin, Z.; Yang, G.; and Belta, C. 2019. A formal methods approach to interpretable reinforcement learning for robotic planning. *Science Robotics*, 4(37): eaay6276.
- Liu, J. X.; Shah, A.; Rosen, E.; Jia, M.; Konidaris, G.; and Tellex, S. 2024. LTL-Transfer: Skill Transfer for Temporal Task Specification. arXiv:2206.05096.
- Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. The MIT Press, second edition.
- Tasse, G. N.; James, S.; and Rosman, B. 2020. A Boolean Task Algebra for Reinforcement Learning. arXiv:2001.01394.
- Tasse, G. N.; Jarvis, D.; James, S.; and Rosman, B. 2024. Skill Machines: Temporal Logic Skill Composition in Reinforcement Learning. arXiv:2205.12532.