Program Synthesis Guided Reinforcement Learning

Yichen Yang [1] **Jeevana Priya Inala** [1] **Osbert Bastani** [2] **Yewen Pu** [3] **Armando Solar-Lezama** [1] **Martin Rinard** [1]

Abstract

A key challenge for reinforcement learning is solving long-horizon planning and control problems. Recent work has proposed leveraging programs to help guide the learning algorithm in these settings. However, these approaches impose a high manual burden on the user since they must provide a guiding program for every new task they seek to achieve. We propose an approach that leverages program synthesis to automatically generate the guiding program. A key challenge is how to handle partially observable environments. We propose *model predictive program synthesis*, which trains a generative model to predict the unobserved portions of the world, and then synthesizes a program based on samples from this model in a way that is robust to its uncertainty. We evaluate our approach on a set of challenging benchmarks, including a 2D Minecraft-inspired "craft" environment where the agent must perform a complex sequence of subtasks to achieve its goal, a box-world environment that requires abstract reasoning, and a variant of the craft environment where the agent is a MuJoCo Ant. Our approach significantly outperforms several baselines, and performs essentially as well as an oracle that is given an effective program.

1. Introduction

Reinforcement learning has been applied to solving challenging planning and control problems (Mnih et al., 2015; Arulkumaran et al., 2017). Despite a significant amount of recent progress, solving long-horizon problems remains a significant challenge due to the combinatorial explosion of possible strategies.

One promising approach to addressing these issues is to leverage *programs* to guide the behavior of the agents (Andreas et al., 2017; Sun et al., 2020). In this paradigm, the user provides a sequence of high-level instructions designed to guide the agent. For instance, the program might encode intermediate subgoals that the agent should aim to achieve, but leave the reinforcement learning algorithm to discover how exactly to achieve these subgoals. In addition, to handle partially observable environments, these programs might encode conditionals that determine the course of action based on the agent's observations.

The primary drawback of these approaches is that the user becomes burdened with providing such a program for every new task. Not only is this process time-consuming for the user, but a poorly written program may hamper learning. A natural question is whether we can automatically *synthesize* these programs. That is, rather than require the user to provide the program, we instead have them provide a high-level specification that encodes only the desired goal. Then, our framework automatically synthesizes a program that achieves this specification. Finally, this program is used to guide the reinforcement learning algorithm.

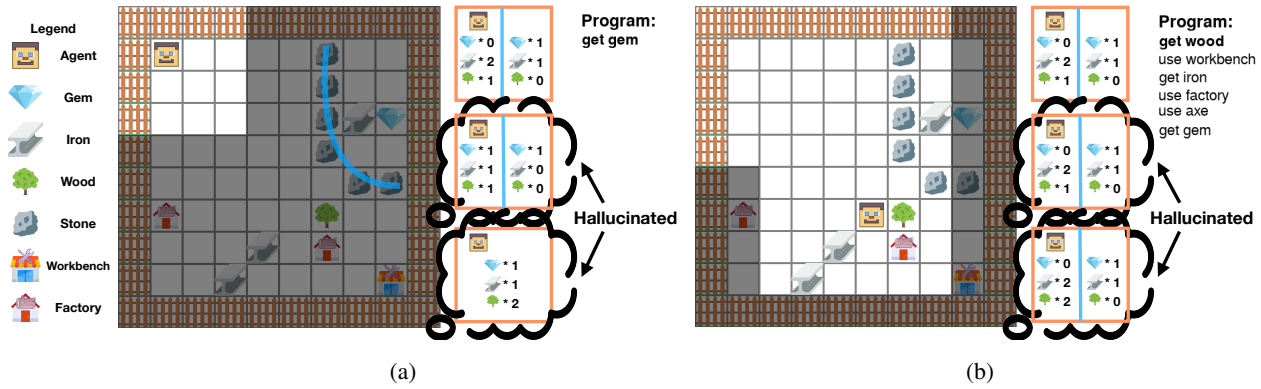
The key challenge to realizing our approach is how to handle partially observable environments. In the fully observed setting, the program synthesis problem reduces to STRIPS planning (Fikes & Nilsson, 1971)—i.e., search over the space of possible plans to find one that achieves the goal. However, these techniques are hard to apply in settings where the environment is initially unknown.

To address this challenge, we propose an approach called *model predictive program synthesis (MPPS)*. At a high level, our approach synthesizes the guiding program based on a conditional generative model of the environment, but in a way that is robust to the uncertainty in this model. In particular, for a user-provided goal specification ϕ, the agent chooses its actions using the following three steps:

- **Hallucinator:** First, inspired by world-models (Ha & Schmidhuber, 2018), the agent keeps track of a conditional generative model g over possible realizations of the unobserved portions of the environment.
- **Synthesizer:** Next, given the world predicted by g, the agent synthesizes a program p that achieves ϕ assuming this prediction is accurate. Since world predictions are stochastic in nature, it samples multiple predicted worlds and computes the program that maximizes the probability of success according to these samples.

[1]MIT EECS & CSAIL [2]University of Pennsylvania [3]Autodesk, Inc.. Correspondence to: Yichen Yang <yicheny@csail.mit.edu>.

Preprint, under review.



(a) (b)

Figure 1. (a) An initial state for the craft environment. Bright regions are observed and dark ones are unobserved. This map has two *zones* separated by a stone boundary (blue line). The first zone contains the agent, 2 irons, and 1 wood; the second contains 1 iron and 1 gem. The goal is to get the gem. The agent represents the high-level structure of the map (e.g., resources in each zone) as *abstraction variables*. The ground truth abstraction variables are in the top-right; we only show the counts of gems, irons, and woods in each zone and the zone containing the agent. The two thought bubbles below are abstract variables hallucinated by the agent based on the observed parts of the map. In both, the zone that the agent is in contains a gem, so the synthesized program is "get gem". However, this program cannot achieve the goal. (b) The state after the agent took 20 actions, failed to obtain the gem, and is now synthesizing a new program. They have explored more of the map, so the hallucinations are more accurate, and the new program is a valid strategy for obtaining the gem.

- **Executor:** Finally, the agent executes the strategy encoded by p for a fixed number of steps N. Concretely, p is a sequence of components $p = c_1; ...; c_k$, where each component is an *option* $c_\tau = (\pi_\tau, \beta_\tau)$ (Sutton et al., 1999), which says to execute policy π_τ until condition β_τ holds.

If ϕ is not satisfied after N steps, then the above process is repeated. Since the hallucinator now has more information (assuming the agent has explored more of the environment), the agent now has a better chance of achieving its goal. Importantly, the agent is implicitly encouraged to explore since it must do so to discover whether the current program can successfully achieve the goal ϕ.

Similar to Sun et al. (2020), the user instantiates our framework in a new domain by providing a set of *prototype components* $\tilde{c}$, where $\tilde{c}$ is a logical formula encoding a useful subtask for that domain. For instance, $\tilde{c}$ may encode that the agent should navigate to a goal position. The user does not need to provide a policy to achieve $\tilde{c}$; our framework uses reinforcement learning to automatically train such a policy c. Our executor reuses these policies c to solve different tasks in varying environments within the same domain. In particular, for a new task and/or environment, the user only needs to provide a specification ϕ, which is a logical formula encoding the goal of that task.

We instantiate this approach in the context of a 2D Minecraft-inspired environment (Andreas et al., 2017; Sohn et al., 2018; Sun et al., 2020), which we call the "craft environment", and a "box-world" environment (Zambaldi et al., 2019). We demonstrate that our approach significantly outperforms existing approaches for partially observable environments, while performing essentially as well as using handcrafted programs to guide the agent. In addition, we demonstrate that the policy we learn can be transferred to a continuous variant of the craft environment, where the agent is replaced by a MuJoCo (Todorov et al., 2012) ant.

Related work. There has been recent interest in program-guided reinforcement learning, where a program encoding high-level instructions on how to achieve the goal (essentially, a sequence of options) is used to guide the agent. Andreas et al. (2017) uses programs to guide agents that are initially unaware of *any* semantics of the programs (i.e., the program is just a sequence of symbols), with the goal of understanding whether the structure of the program alone is sufficient to improve learning. Jothimurugan et al. (2019) enables users to write specifications in a high-level language based on temporal logic. Then, they show how to translate these specifications into shaped reward functions to guide learning. Most closely related is recent work (Sun et al., 2020) that has demonstrated how program semantics can be used to guide reinforcement learning in the craft environment. As with this work, we assume that the user provides semantics of each option in the program (i.e., the subgoal that should be achieved by that option), but not an actual policy implementing this option (which is learned using reinforcement learning). However, we do not assume that the user provides the program, just the overall goal.

More broadly, our work fits into the literature on combining high-level planning with reinforcement learning. In particular, there is a long literature on planning with options (Sutton et al., 1999) (also known as *skills* (Hausman et al., 2018)), including work on inferring options (Stolle & Precup, 2002).

However, these approaches cannot be applied to MDPs with continuous state and action spaces or to partially observed MDPs. Recent work has addressed the former (Abel et al., 2020; Jothimurugan et al., 2021) by combining high-level planning with reinforcement learning to handle low-level control, but not the latter, whereas our work tackles both challenges. Similarly, classical planning algorithms such as STRIPS (Fikes & Nilsson, 1971) cannot handle uncertainty in the realization of the environment. There has also been work on *replanning* (Stentz et al., 1995) to handle small changes to an initially known environment, but they cannot handle environments that are initially completely unknown. Alternatively, there has been work on hierarchical planning in POMDPs (Charlin et al., 2007; Toussaint et al., 2008), but these are not designed to handle continuous state and action spaces. We leverage program synthesis (Solar-Lezama, 2008) in conjunction with the world models approach (Ha & Schmidhuber, 2018) to address these issues.

Finally, there has broadly been recent interest in using program synthesis to learn programmatic policies that are more interpretable (Verma et al., 2018; Inala et al., 2021), verifiable (Bastani et al., 2018; Verma, 2019), and generalizable (Inala et al., 2020). In contrast, we are not directly synthesizing the policy, but a program to guide the policy.

2. Motivating Example

Figure 1a shows a 2D Minecraft-inspired crafting game. In this grid world, the agent can navigate and collect resources (e.g., wood), build tools (e.g., a bridge) at workshops using collected resources, and use the tools to achieve subtasks (e.g., use a bridge to cross water). The agent can only observe the 5×5 grid around its current position; since the environment is static, it also memorizes locations it has seen before. A single task consists of a randomly generated map (i.e., the environment) and goal (i.e., obtain a certain resource or build a certain tool).

To instantiate our framework, we provide prototype components that specify high-level behaviours such as getting wood or using toolshed to build a bridge. Figure 2 shows the domain-specific language that encodes the set of prototypes.

For each prototype, we need to provide a logical formula $\tilde{c}$ that formally specifies its desired behavior. Rather than specifying behavior over concrete state s, we instead specify it over *abstraction variables* that encode subsets of the state space. For instance, we divide the map into *zones* that are regions separated by obstacles such as water and stone. As an example, the map in Figure 1a has two zones: the region containing the agent and the region blocked off by stones. Then, the zone the agent is currently in is represented by an abstraction variable z—i.e., the states s where the agent is in zone i is represented by the logical predicate $z = i$.

The prototype components are logical formulas over these abstraction variables—e.g., the prototype for "get wood" is

$$\forall i, j \,.\, (z^- = i \wedge z^+ = j) \Rightarrow (b^-_{i,j} = \text{connected}) \\ \wedge (\rho^+_{j,\text{wood}} = \rho^-_{j,\text{wood}} - 1) \wedge (\iota^+_{\text{wood}} = \iota^-_{\text{wood}} + 1).$$

In this formula, $b_{i,j}$ indicates whether zones i and j are connected, $\rho_{i,r}$ denotes the count of resource r in zone i, and ι_r denotes the count of resource r in the agent's inventory. The $+$ and $-$ superscripts on each abstraction variable indicates that it represents the initial state of the agent before the execution of the prototype and the final state of the agent after the execution of the prototype, respectively.

Thus, this formula says that (i) the agent goes from zone i to j, (ii) i and j are connected, (iii) the count of wood in the agent's inventory increases by one, and (iv) the count of wood in zone j decreases by one. All of the prototype components we use are summarized in Appendix A.

Before solving any tasks, for each prototype $\tilde{c}$, our framework uses reinforcement learning to train a component c that implements $\tilde{c}$—i.e., an option $c = (\pi, \beta)$ that attempts to satisfy the behavior encoded by the logical formula $\tilde{c}$.

To solve a new task, the user provides a logical formula ϕ encoding the goal of this task. Then, the agent acts in the environment to try achieve ϕ. For example, Figure 1a shows the initial state of an agent where the task is to obtain a gem.

First, based on the observations so far, the agent π uses the hallucinator g to predict multiple potential worlds, each of which represents a possible realization of the full map. One convenient aspect of our approach is that rather than predicting concrete states, it suffices to predict the abstraction variables used in the prototype components $\tilde{c}$ and goal specification ϕ. For instance, Figure 1a shows two samples of the world predicted by g; here, the only values it predicts are the number of zones in the map, the type of the boundary between the zones, and the counts of the resources and workshops in each zone. In this example, the first predicted world contains two zones, and the second contains one zone. Note that in both predicted worlds, there is a gem located in same zone as the agent.

Next, π synthesizes a program p that achieves the goal in the maximum possible number of predicted worlds. The synthesized program in Figure 1a is a single component "get gem", which is an option that searches the current zone (or zones already connected with the current zone) for a gem. Note that this program achieves the goal for the predicted worlds shown in Figure 1a.

Finally, the agent executes the program $p = c_1; ...; c_k$ for a fixed number N of steps. In particular, it executes the policy π_τ of component $c_\tau = (\pi_\tau, \beta_\tau)$ until β_τ holds, upon which it switches to executing $c_{\tau+1}$. In our example, there is only

$$
\begin{array}{rcl}
C & := & \text{get } R \mid \text{use } T \mid \text{use } W \\
R & := & \text{wood} \mid \text{iron} \mid \text{grass} \mid \text{gold} \mid \text{gem} \\
T & := & \text{bridge} \mid \text{axe} \\
W & := & \text{factory} \mid \text{workbench} \mid \text{toolshed}
\end{array}
$$

Figure 2. Prototype components for the craft environment; the three kinds of prototypes are get resource (R), use tool (T), and use workshop (W).

one component "get gem", so it executes the policy for this component until the agent finds a gem.

In this case, the agent fails to achieve ϕ since there is no gem in the same zone as the agent. Thus, the agent repeats the above process. Since the agent now has more observations, g more accurately predicts the world. For instance, Figure 1b shows the intermediate step when the agent does the first replanning. Note that it now correctly predicts that the only gem is in the second zone. As a result, the newly synthesized program is

$$
p = \overbrace{\text{get wood; use workbench; get iron; use factory;}}^{\text{for building axe}} \\
\text{use axe; get gem.}
$$

That is, it builds an axe to break the stone so it can get to the zone containing the gem. Finally, the agent executes this new program, which successfully finds the gem.

3. Problem Formulation

POMDP. We consider a partially observed Markov decision process (POMDP) with states $\mathcal{S} \subseteq \mathbb{R}^n$, actions $\mathcal{A} \subseteq \mathbb{R}^m$, observations $\mathcal{O} \subseteq \mathbb{R}^q$, initial state distribution $\mathcal{P}_0$, observation function $h : \mathcal{S} \to \mathcal{O}$, and transition function $f : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Given initial state $s_0 \sim \mathcal{P}_0$, policy $\pi : \mathcal{S} \to \mathcal{A}$, and time horizon $T \in \mathbb{N}$, the generated trajectory is $(s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$, where $o_t = h(s_t)$, $a_t = \pi(o_t)$, and $s_{t+1} = f(s_t, a_t)$.

We assume that the state includes the unobserved parts of the environment—e.g., in our craft environment, it represents both the entire map as well as the agent's current position.

Programs. We consider programs $p = c_1; ...; c_k$ that are composed of *components* $c_\tau \in C$. Each component c represents an option $c = (\pi, \beta)$, where $\pi : \mathcal{O} \to \mathcal{A}$ is a policy and $\beta : \mathcal{O} \to \{0, 1\}$. To execute p, the agent uses the options $c_1, ..., c_k$ in sequence. To use option $c_\tau = (\pi_\tau, \beta_\tau)$, it takes actions $\pi_\tau(o)$ until $\beta_\tau(o) = 1$; at this point, the agent switches to option $c_{\tau+1}$ and continues this process.

User-provided prototype components. Rather than have the user directly provide the components C used in our programs, we instead have them provide *prototype components* $\tilde{c} \in \tilde{C}$. Importantly, prototypes can be shared across closely related tasks. Each prototype component is a logical formula that encodes the expected desired behavior of a component. More precisely, $\tilde{c}$ is a logical formula over variables s^- and s^+, where s^- denotes the initial state before executing the option and s^+ denotes the final state after executing the option. For instance, the prototype component

$$\tilde{c} \equiv (s^- = s_0 \Rightarrow s^+ = s_1) \vee (s^- = s_2 \Rightarrow s^+ = s_3)$$

says that if the POMDP is currently in state s_0, then c should transition it to s_1, and if it is currently in state s_2, then c should transition it to s_3.

Rather than directly define $\tilde{c}$ over the states s, we can instead define it over *abstraction variables* that represent subsets of the state space. This approach can improve scalability of our synthesis algorithm—e.g., it enables us to operate over continuous state spaces as long as the abstraction variables themselves are discrete.

User-provided specification. To specify a task, the user provides a specification ϕ, which is a logical formula over states s; in general, ϕ may not directly refer to s but to other variables that represent subsets of $\mathcal{S}$. Our goal is to design an agent that achieves any given ϕ (i.e., act in the POMDP to reach a state that satisfies ϕ) as quickly as possible.

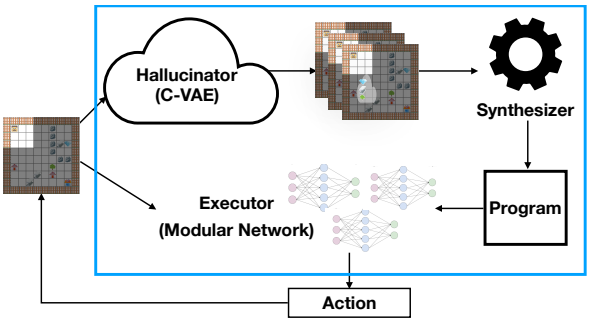


Figure 3. Architecture of our agent (the blue box).

4. Model Predictive Program Synthesis

We describe the architecture of our agent, depicted in Figure 3. It is composed of three parts: the *hallucinator* g, which predicts possible worlds; the *synthesizer*, which generates a program p that succeeds with high probability according to worlds sampled from g; and the *executor*, which uses p to act in the POMDP. These parts are run once every N steps to generate a program p to execute for the subsequent N steps, until the user-provided specification ϕ is achieved.

Hallucinator. First, the hallucinator is a conditional generative model trained to predict the environment given the observation so far. For simplicity, we assume the observation o on the current step already encodes all observations

so far. Since our craft environment is static, o simply encodes the portion of the map that has been revealed so far, with a special symbol indicating parts that are unknown. To be precise, the hallucinator g encodes a distribution $g(s \mid o)$, which is trained to approximate the actual distribution $P(s \mid o)$. Then, at each iteration (i.e., once every N steps), our agent samples m worlds $\hat{s}_1, ..., \hat{s}_m \sim g(\cdot \mid o)$. We choose g to be a conditional variational auto-encoder (CVAE) (Sohn et al., 2015).

When using abstract variables to represent the states, we can have g directly predict the values of these abstract variables instead of having g predict the concrete state. Intuitively, this approach works since as described below, the synthesizer only needs to know the values of the abstract variables to generate a program.

Synthesizer. The synthesizer aims to compute the program that maximizes the probability of satisfying the goal ϕ:

$$\begin{aligned} p^* &= \arg\max_p \mathbb{E}_{P(s|o)}[p \text{ solves } \phi \text{ for } s] \\ &\approx \arg\max_p \frac{1}{m}\sum_{j=1}^{m} \mathbb{1}[p \text{ solves } \phi \text{ for } \hat{s}_j], \end{aligned} \tag{1}$$

where the $\hat{s}_j$ are samples from g. The objective (1) can be expressed as a MaxSAT problem (Krentel, 1986). In particular, suppose for now that we are searching over programs $p = c_1; ...; c_k$ of fixed length k. Then, consider the constrained optimization problem

$$\arg\max_{\xi_1,...,\xi_k} \frac{1}{m}\sum_{j=1}^{m} \exists s_1^-, s_1^+, ..., s_k^-, s_k^+ \;.\; \psi_j, \tag{2}$$

where ξ_τ and s_τ^δ (for $\tau \in \{1, ..., k\}$ and $\delta \in \{-, +\}$) are the optimization variables. Intuitively, $\xi_1, ..., \xi_k$ encodes the program $p = c_1; ...; c_k$, and ψ_j encodes the event that p solves ϕ for world $\hat{s}_j$. In particular, we have

$$\psi_j \equiv \psi_{j,\text{start}} \wedge \left[\bigwedge_{\tau=1}^{k} \psi_{j,\tau}\right] \wedge \left[\bigwedge_{\tau=1}^{k-1} \psi'_{j,\tau}\right] \wedge \psi_{j,\text{end}},$$

where

$$\psi_{j,\text{start}} \equiv (s_1^- = \hat{s}_j)$$

encodes that the initial state is $\hat{s}_j$,

$$\psi_{j,\tau} \equiv \big((\xi_\tau = \tilde{c}) \Rightarrow \tilde{c}(s_\tau^-, s_\tau^+)\big)$$

encodes that if the τth component has prototype $\tilde{c}$, then the τth component should transition the system from s_τ^- to s_τ^+,

$$\psi'_{j,\tau} \equiv (s_\tau^+ = s_{\tau+1}^-)$$

encodes that the final state of component τ should equal the initial state of component $\tau + 1$, and

$$\psi_{j,\text{end}} \equiv \phi(s_j^+)$$

encodes that the final state of the last component should satisfy the user-provided goal ϕ.

We use a MaxSAT solver to solve (2) (De Moura & Bjørner, 2008). Given a solution $\xi_1 = \tilde{c}_1, ..., \xi_k = \tilde{c}_k$, the synthesizer returns the corresponding program $p = c_1; ...; c_k$.

We incrementally search for longer and longer programs, starting from $k = 1$ and incrementing k until either we find a program that achieves at least a minimum objective value, or we reach a maximum program length $k_{\max}$, at which point we use the best program found so far.

Executor. The executor runs the synthesized program $p = c_1; ...; c_k$ for the subsequent N steps. It iteratively uses each component $c_\tau = (\pi_\tau, \beta_\tau)$, starting from $\tau = 1$. In particular, it uses action $a_t = \pi_\tau(o_t)$ at each time step t, where o_t is the observation on that step. It does so until $\beta_\tau(o_t) = 1$, at which point it increments $\tau \leftarrow \tau + 1$.

Finally, it continues until either it has completed running the program (i.e., $\beta_k(o_t) = 1$), or after N time steps. In the former case, by construction, the goal ϕ has been achieved, so the agent terminates. In the latter case, the agent iteratively reruns the above three steps based on the current observation to synthesize a new program. At this point, the hallucinator likely has additional information about the environment, so the new program has a greater chance of achieving ϕ.

5. Learning Algorithm

Next, we describe our algorithm for learning the parameters of models used by our agent. In particular, there are two parts that need to be learned: (i) we need to learn parameters of the conditional variational auto-encoder (CVAE) hallucinator g, and (ii) we need to learn the components c based on the user-provided prototype components $\tilde{c}$.

Hallucinator. We choose the hallucinator g to be a conditional variational auto-encoder (CVAE) (Sohn et al., 2015) trained to estimate the distribution $P(s \mid o)$ of states given the current observation. First, we obtain samples (o_t, s_t) using rollouts collected using a random agent. Then, we train the CVAE using the standard evidence lower bound (ELBo) on the log likelihood (Kingma & Welling, 2013):

$$\begin{aligned} \ell(\theta, \tilde{\theta}) = \mathbb{E}_{P(s,o)}\Big[&\mathbb{E}_{h_{\tilde{\theta}}(z|s,o)}[\log g_\theta(s \mid z, o)] \\ &- D_{\text{KL}}(h_{\tilde{\theta}}(z \mid s, o) \parallel \mathcal{N}(z; 0, 1))\Big], \end{aligned} \tag{3}$$

where $h_{\tilde{\theta}}$ is the encoder and g_θ is the decoder:

$$\begin{aligned} h_{\tilde{\theta}}(z \mid s, o) &= \mathcal{N}(z; \tilde{\mu}_{\tilde{\theta}}(s, o), \tilde{\sigma}_{\tilde{\theta}}(s, o)^2 \cdot I) \\ g_\theta(s \mid z, o) &= \mathcal{N}(s; \mu_\theta(z, o), \sigma_\theta(z, o)^2 \cdot I), \end{aligned}$$

where μ_θ, σ_θ, $\tilde{\mu}_{\tilde{\theta}}$, and $\tilde{\sigma}_{\tilde{\theta}}$ are neural networks, and I is the identity matrix. We train $h_{\tilde{\theta}}$ and g_θ by jointly optimizing (3), and then choose the hallucinator to be $g = g_\theta$.

Executor. Our framework uses reinforcement learning to learn components c that implement the user-provided prototype components $\tilde{c}$. The learned components $c \in C$ can be shared across multiple tasks. Our approach is based on neural module networks for reinforcement learning Andreas et al. (2017). In particular, we train a neural module π for each component c. In addition, we construct a monitor β that checks when to terminate execution, and take $c = (\pi, \beta)$.

First, β is constructed from $\tilde{c}$—in particular, it returns whether $\tilde{c}$ is satisfied based on the current observation o. Note that we have assumed that $\tilde{c}$ can be checked based only on o; this assumption holds for all prototypes in our craft environment. If it does not hold, we additionally train π to explore in a way that enables it to check $\tilde{c}$.

Now, to train the policies π, we generate random initial states s and goal specifications ϕ. For training, we use programs synthesized from the fully observed environments; such a program p is guaranteed to achieve ϕ from s. We use this approach since it avoids the need to run the synthesizer repeatedly during training.

Then, we sample a rollout $\{((o_1, a_1, r_1), ..., (o_T, a_T, r_T))\}$ by using the executor in conjunction with the program p and the current options $c_\tau = (\pi_\tau, \beta_\tau)$ (where π_τ is randomly initialized). We give the agent a reward $\tilde{r}$ on each time step where achieves the subgoal of a single component c_τ—i.e., the executor increments $\tau \leftarrow \tau + 1$. Then, we use actor-critic reinforcement learning (Konda & Tsitsiklis, 2000) to update the parameters of each policy π.

Finally, as in Andreas et al. (2017), we use curriculum learning to speed up training—i.e., we train using goals that can be achieved with shorter programs first.

6. Experiments

In this section, we describe empirical evaluations of our approach. As we show, it significantly outperforms non-program-guided baselines, while performing essentially as well as an oracle that is given the ground truth program.

6.1. Benchmarks

2D-craft. We consider a 2D Minecraft-inspired crafting game based on the ones in Andreas et al. (2017); Sun et al. (2020) (Figure 1a). A map in this domain is an 8×8 grid, where each grid cell either is empty or contains a resource (e.g., wood or gold), an obstacle (e.g., water or stone), or a workshop. In each episode, we randomly sample a map from a predefined distribution, a random initial position for the agent, and a random task (one of 14 possibilities, each of which involves getting a certain resource or building a certain tool). The more complicated tasks may require the agent to build intermediate tools (e.g., a bridge or an axe) to reach initially inaccessible regions to achieve its goal. In contrast to prior work, our agent does not initially observe the entire map; instead, they can only observe grid cells in a 5×5 square around them. Since the environment is static, any previously visited cells remain visible. The agent has a discrete action space, including move actions in four directions, and a special "use" action that can pick a resource, use a workshop, or use a tool. The maximum length of each episode $T = 100$.

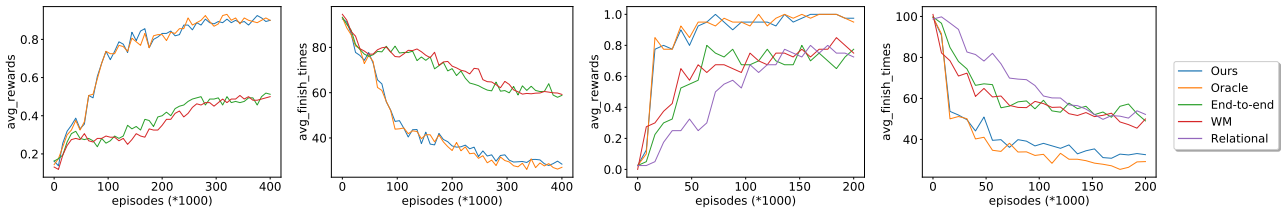
Ant-craft. Next, we consider a variant of 2D-craft where the agent is replaced with a MuJoCo (Todorov et al., 2012) ant (Schulman et al., 2016) (illustrated in Figure 5a). For simplicity, we do not model the physics of the interaction between the ant and its environment—e.g., the ant automatically picks up resources in the grid cell it currently occupies. The policy needs to learn the continuous control to walk the ant as well as the strategy to perform the tasks. This environment is designed to demonstrate that our approach can be applied to continuous control tasks.

Box-world. Finally, we consider the box-world environment (Zambaldi et al., 2019), which requires abstract relational reasoning. It is a 12×12 grid world with locks and boxes randomly scattered throughout (visualized in Figure 5b). Each lock occupies a single grid cell, and the box it locks occupies the adjacent grid cell. The box contains a key that can open a subsequent lock. Each lock and box is colored; the key needed to open a lock is contained in the box of the same color. The agent is given a key to get started, and its goal is to unlock the box of a given color. The agent can move in the room in four directions; it opens a lock for which it has the key simply by walking over it, at which point it can pick up the adjacent key. We assume that once the agent has the key of a given color, it can unlock multiple locks of that color. We modify the original environment to be partially observable; in particular, the agent can observe a 7×7 grid around them (as well as the previously observed grid cells). In each episode, we sample a random configuration of the map, where the number of boxes in the path to the goal is randomly chosen between 1 to 4, and the number of "distractor branches" (i.e., boxes that the agent can open but does not help them reach the goal) is also randomly chosen between 1 to 4.

6.2. Baselines

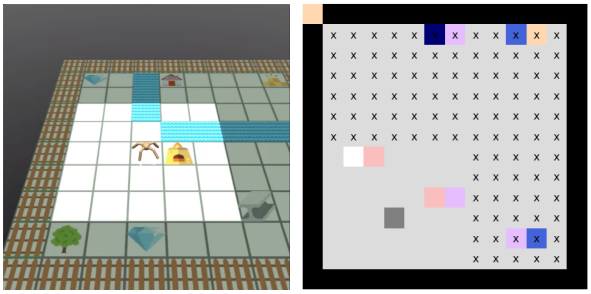
End-to-end. An end-to-end neural policy trained with the same actor-critic algorithm and curriculum learning as discussed in Section 5. It uses one actor network per task.

World models. The world models approach (Ha & Schmidhuber, 2018) handles partial observability by using a generative model to predict the future. It trains a V model (VAE) and an M model (MDN-RNN) to learn a compressed spa-



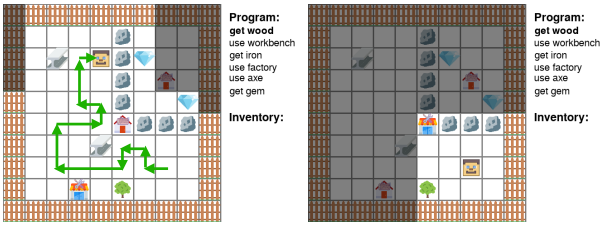
(a) (b) (c) (d)

Figure 4. (a,b) Training curves for 2D-craft environment. (c,d) Training curves for the box-world environment. (a,c) The average reward on the test set over the course of training; the agent gets a reward of 1 if it successfully finishes the task in the time horizon, and 0 otherwise. (b,d) The average number of steps taken to complete the task on the test set. We show our approach ("Ours"), the program guided agent ("Oracle"), the end-to-end neural policy ("End-to-end"), world models ("World models"), and relational deep RL ("Relational").



(a) (b)

Figure 5. (a) The Ant-craft environment. The policy needs to control the ant to perform the crafting tasks. (b) The box-world environment. The grey pixel denotes the agent. The goal is to get the white key. The unobserved parts of the map is marked with "x". The key currently held by the agent is shown in the top-left corner. In this map, the number of boxes in the path to the goal is 4, and it contains 1 distractor branch.



(a) (b)

Figure 6. Comparison of behaviors between the optimistic approach (left) and our MPPS approach (right), in a scenario where goal is to get the gem. (a) This state is the point at which the optimistic approach first synthesizes the correct program instead of the (incorrect) one "get gem". It only does so after the agent has observed all the squares in its current zone (the green arrows show the agent's trajectory so far). (b) The initial state of our MPPS strategy. It directly synthesizes the correct program, since the hallucinator knows the gem is most likely in the other zone. Thus, the agent completes the task much more quickly.

tial and temporal representation of the environment. The V model takes the observations at each step t and encodes it into a latent vector z_t. The M model is a recurrent model that takes the latent vectors $z_1, ..., z_t$ as input and predicts z_{t+1}. The latent states of the M model and the latent vectors z_t from the V model together form the world model features, which are used as inputs to the controller (C model).

Program guided agent. This technique uses a program to guide the agent policy (Sun et al., 2020). Unlike our approach, the ground truth programs (i.e., a program guaranteed to achieve the goal) is provided to the agent at the beginning; we synthesize this program using the full map (i.e., including parts of the map that are unobserved by the agent). This baseline can be viewed as an oracle since it is strictly more powerful than our approach.

Relational Deep RL. For the box-world environment, we also compare with the relational deep RL approach (Zambaldi et al., 2019), which replaces the policy network with a relational module based on the multi-head attention mechanism (Vaswani et al., 2017) operating over the map features. The output of the relational module is used as input to an MLP network that computes the action.

6.3. Implementation Details

2D-craft environment. For our approach, we use a CVAE as the hallucinator with MLPs (a hidden layer of dimension 200) for the encoder and the decoder. We pre-train the CVAE on 100 rollouts with 100 timesteps in each rollout—i.e., 10,000 (s, o) pairs. We use the Z3 SMT solver (De Moura & Bjørner, 2008) to solve the MAXSAT synthesis formula. We set the number of sample completions $m = 3$, and the number of steps to replan $N = 20$. We use the same architecture for the actor networks and critic networks across our approach and all baselines: for actor networks, we use MLP with a hidden layer of dimension 128, and for critic networks, we use MLP with a hidden layer of dimension 32. We train each model on 400K episodes, and evaluate on a test set containing 10 scenarios per task.

Ant-craft. We first pre-train a goal following policy for the ant: given a randomly chosen goal position, this policy con-

	Avg. reward	Avg. finish step
End-to-end	0.49	60.7
World models	0.50	59.3
Ours	**0.93**	26.7
Oracle	**0.93**	**25.9**

Table 1. Average rewards and average completion times (i.e., number of steps) on the test set for Ant-craft environment, for the best policy found for each approach.

trols the ant to move to that position. We use the soft actor-critic algorithm (Haarnoja et al., 2018) for pre-training. The executor in our approach, as well as our baseline policies, outputs actions that are translated into the goal positions as inputs to this ant controller. We let the ant controller run for 50 timesteps in the simulator to execute each move action from the upper-stream policies. We initialize each policy with the trained model from the 2D-craft environment, and fine-tune it on the Ant-craft environment for 40K episodes.

Box-world. Following Zambaldi et al. (2019), we use a one-layer CNN with 32 kernels of size 3×3 to process the raw map inputs before feeding into the downstream networks across all approaches. For the programs in our approach, we have a prototype component for each color, where the desired behavior of the component is to get the key of that color. The full definition of the prototype components we use for box-world is in Appendix B. For the hallucinator CVAE, we use the same architecture as in the craft environment with a hidden dimension of 300, and trained with 100k (s, o) pairs. For the synthesizer, we set $m = 3$ and $N = 10$. We train each model for 200K episodes, and evaluate on a test set containing 10 scenarios per level. Each level has a specific number of boxes in the path to the goal (i.e., the goal length). Our test set contains four levels with goal lengths between 1 to 4.

6.4. Results

2D-craft. Figures 4a & 4b show the training curves of each approach. As can be seen, our approach learns a substantially better policy than the unsupervised baselines; it solves a larger percentage of test scenarios as well as using shorter time. Compared with program guided agent (i.e., the oracle), our approach achieves a similar average reward with slightly longer average finish time. These results demonstrate that our approach significantly outperforms non-program-guided baselines, while performing nearly as well as an oracle that knows the ground truth program.

Ant-craft. Table 1 shows results for the best policy found using each approach. As before, our approach significantly outperforms the baseline approaches while performing comparably with the oracle approach.

Box-world. Figure 4c & 4d shows the training curves. As before, our approach performs substantially better than the baselines, and achieves a similar performance as the program guided agent (i.e., the oracle).

	Avg. reward	Avg. finish step
Optimistic	0.60	53.7
Ours	**0.79**	41.8
Oracle	**0.79**	**37.7**

Table 2. Comparison to optimistic ablation on challenging tasks for the 2D-craft environment.

6.5. Optimistic Ablation

Finally, we compare our model predictive program synthesis with an alternative, *optimistic* synthesis strategy: it considers the unobserved parts of the map to be possibly in any configurations, and synthesizes the shortest program as long as it works on any of these possibilities. We compare on the most challenging tasks for 2D-craft (i.e., get gold or get gem), since for these tasks, the ground truth program depends heavily on the map. We show results in Table 2. As can be seen, our approach significantly outperforms the optimistic synthesis approach, and performs comparably to the oracle. Finally, in Figure 6, we illustrate the difference in behavior between our approach and the optimistic strategy.

7. Conclusion

We have proposed an algorithm for synthesizing programs to guide reinforcement learning. Our algorithm, called model predictive program synthesis, handles partially observed environments by leveraging the world models approach, where it learns a generative model over the remainder of the world conditioned on the observations thus far. In particular, it synthesizes a guiding program that accounts for the uncertainty in the world model. Our experiments demonstrate that our approach significantly outperforms non-program-guided approaches, while performing comparably to an oracle that is given access to the ground truth program. These results demonstrate that our approach can obtain the benefits of program-guided reinforcement learning without requiring the user to provide a guiding program for every new task and world configurations.

References

Abel, D., Umbanhowar, N., Khetarpal, K., Arumugam, D., Precup, D., and Littman, M. Value preserving state-action abstractions. In *International Conference on Artificial Intelligence and Statistics*, pp. 1639–1650. PMLR, 2020.

Andreas, J., Klein, D., and Levine, S. Modular multitask reinforcement learning with policy sketches. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th Inter-*

national Conference on Machine Learning, volume 70 of *Proceedings of Machine Learning Research*, pp. 166–175, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL http://proceedings.mlr.press/v70/andreas17a.html.

Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

Bastani, O., Pu, Y., and Solar-Lezama, A. Verifiable reinforcement learning via policy extraction. *arXiv preprint arXiv:1805.08328*, 2018.

Charlin, L., Poupart, P., and Shioda, R. Automated hierarchy discovery for planning in partially observable environments. *Advances in Neural Information Processing Systems*, 19:225, 2007.

De Moura, L. and Bjørner, N. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pp. 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787992.

Fikes, R. E. and Nilsson, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

Ha, D. and Schmidhuber, J. World models. *CoRR*, abs/1803.10122, 2018. URL http://arxiv.org/abs/1803.10122.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1861–1870, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL http://proceedings.mlr.press/v80/haarnoja18b.html.

Hausman, K., Springenberg, J. T., Wang, Z., Heess, N., and Riedmiller, M. Learning an embedding space for transferable robot skills. In *International Conference on Learning Representations*, 2018.

Inala, J. P., Bastani, O., Tavares, Z., and Solar-Lezama, A. Synthesizing programmatic policies that inductively generalize. In *International Conference on Learning Representations*, 2020.

Inala, J. P., Yang, Y., Paulos, J., Pu, Y., Bastani, O., Kumar, V., Rinard, M., and Solar-Lezama, A. Neurosymbolic transformers for multi-agent communication. *arXiv preprint arXiv:2101.03238*, 2021.

Jothimurugan, K., Alur, R., and Bastani, O. A composable specification language for reinforcement learning tasks. In *NeurIPS*, 2019.

Jothimurugan, K., Bastani, O., and Alur, R. Abstract value iteration for hierarchical reinforcement learning. In *AISTATS*, 2021.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Konda, V. and Tsitsiklis, J. Actor-critic algorithms. In Solla, S., Leen, T., and Müller, K. (eds.), *Advances in Neural Information Processing Systems*, volume 12, pp. 1008–1014. MIT Press, 2000. URL https://proceedings.neurips.cc/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

Krentel, M. W. The complexity of optimization problems. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pp. 69–76, New York, NY, USA, 1986. Association for Computing Machinery. ISBN 0897911938. doi: 10.1145/12130.12138. URL https://doi.org/10.1145/12130.12138.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

Sohn, K., Lee, H., and Yan, X. Learning structured output representation using deep conditional generative models. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 28, pp. 3483–3491. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper/2015/file/8d55a249e6baa5c06772297520da2051-Paper.pdf.

Sohn, S., Oh, J., and Lee, H. Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pp. 7156–7166, Red Hook, NY, USA, 2018. Curran Associates Inc.

Solar-Lezama, A. *Program synthesis by sketching*. Citeseer, 2008.

Stentz, A. et al. The focussed dˆ* algorithm for real-time replanning. In *IJCAI*, volume 95, pp. 1652–1659, 1995.

Stolle, M. and Precup, D. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pp. 212–223. Springer, 2002.

Sun, S.-H., Wu, T.-L., and Lim, J. J. Program guided agent. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=BkxUvnEYDH.

Sutton, R. S., Precup, D., and Singh, S. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2): 181–211, 1999.

Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109.

Toussaint, M., Charlin, L., and Poupart, P. Hierarchical pomdp controller optimization by likelihood maximization. In *UAI*, volume 24, pp. 562–570, 2008.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2017.

Verma, A. Verifiable and interpretable reinforcement learning through program synthesis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 9902–9903, 2019.

Verma, A., Murali, V., Singh, R., Kohli, P., and Chaudhuri, S. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pp. 5045–5054. PMLR, 2018.

Zambaldi, V., Raposo, D., Santoro, A., Bapst, V., Li, Y., Babuschkin, I., Tuyls, K., Reichert, D., Lillicrap, T., Lockhart, E., Shanahan, M., Langston, V., Pascanu, R., Botvinick, M., Vinyals, O., and Battaglia, P. Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HkxaFoC9KQ.

A. Prototype Components for Craft

In this section, we describe the prototype components (i.e., logical formulas encoding option pre/postconditions) that we use for the craft environment. First, recall that the domain-specific language that encodes the set of prototypes for the craft environment is

$$\begin{array}{rcl} C & := & \text{get } R \mid \text{use } T \mid \text{use } W \\ R & := & \text{wood} \mid \text{iron} \mid \text{grass} \mid \text{gold} \mid \text{gem} \\ T & := & \text{bridge} \mid \text{axe} \\ W & := & \text{factory} \mid \text{workbench} \mid \text{toolshed} \end{array}$$

Also, the set of possible artifacts (objects that can be made in some workshop using resources or other artifacts) in the craft environment is

$$A = \left\{ \begin{array}{c} \text{bridge, axe, plank, stick, cloth,} \\ \text{rope, bed, shears, ladder} \end{array} \right\}.$$

We define the following abstraction variables:

- **Zone:** $z = i$ indicates the agent is in zone i
- **Boundary:** $b_{i,j} = b$ indicates how zones i and j are connected, where

$$b \in \{\text{connected}, \text{water}, \text{stone}, \text{not adjacent}\}$$

- **Resource:** $\rho_{i,r} = n$ indicates that there are n units of resource r in zone i
- **Workshop:** $\omega_{i,r} = b$, where $b \in \{\text{true}, \text{false}\}$, indicates whether there exists a workshop r in zone i
- **Inventory:** $\iota_r = n$ indicates that there are n objects r (either a resource or an artifact) in the agent's inventory

We use $z^-, b^-, \rho^-, \omega^-, \iota^-$ and $z^+, b^+, \rho^+, \omega^+, \iota^+$ to denote the initial state and final state for a prototype components, respectively. Now, the logical formulae for each prototype components are defined as follows.

(1) "get r" (for any resource $r \in R$). First, we have the following prototype component telling the agent to obtain a specific resource r:

$$\begin{aligned} \forall i, j \,.\, (z^- = i \wedge z^+ = j) \Rightarrow (b^-_{i,j} = \text{connected}) \\ \wedge\, (\rho^+_{j,r} = \rho^-_{j,r} - 1) \wedge (\iota^+_r = \iota^-_r + 1) \wedge \mathcal{Q}. \end{aligned}$$

Here, $\mathcal{Q}$ refers to the conditions that the other fields of the abstract state stay the same—i.e.,

$$\begin{aligned} (b^+ = b^-) \wedge (\omega^+ = \omega^-) \wedge (\iota^+_{\setminus r} = \iota^-_{\setminus r}) \\ \wedge\, (\rho^+_{\setminus (j,r)} = \rho^-_{\setminus (j,r)}), \end{aligned}$$

where $\iota_{\setminus r}$ means all the other fields in ι except ι_r, and similarly for $\rho_{\setminus (j,r)}$. In particular $\mathcal{Q}$ addresses the *frame problem* from classical planning.

(2) "use r" (for any workshop $r \in W$). Next, we have a prototype component telling the agent to use a workshop to create an artifact. To do so, we introduce a set of auxiliary variables to denote the number of artifacts made in this component: $m_o = n$ indicates that n units of artifact o is made, the set of artifacts that can be made at workshop r as A_r, and the number of units of ingredient q needed to make 1 unit of artifact o as $k_{o,q}$, where $q \in R \cup A$; note that $\{A_r\}$ and $\{k_{o,q}\}$ come from the rule of the game.

Then, the logical formula for "use r" is

$$\begin{aligned} &\forall i, j \,.\, (z^- = i \wedge z^+ = j) \Rightarrow (b^-_{i,j} = \text{connected}) \\ &\quad \wedge (w_{j,r} = \text{true}) \wedge \left(\sum_{o \in A_r} m_o \geq 1 \right) \wedge \left(\sum_{o \notin A_r} m_o = 0 \right) \\ &\quad \wedge \left(\forall q \in R,\ \iota^+_q = \iota^-_q - \sum_{o \in A_r} k_{o,q} m_o \right) \\ &\quad \wedge \left(\forall q \in A,\ \iota^+_q = \iota^-_q - \sum_{o \in A_r} k_{o,q} m_o + m_q \right) \\ &\quad \wedge \left(\forall o \in A_r,\ \neg \left(\bigwedge_q \iota^+_q \geq k_{o,q} \right) \right) \\ &\quad \wedge \mathcal{Q}, \end{aligned}$$

where

$$\mathcal{Q} = (b^+ = b^-) \wedge (\omega^+ = \omega^-) \wedge (\rho^+ = \rho^-).$$

This formula reflects the game setting that when the agent uses a workshop, it will make the artifacts until the ingredients in the inventory are depleted.

(3) "use r" ($r =$ bridge/axe). Next, we have the following prototype component for telling the agent to use a tool. The formula for this prototype component encodes the logic of zone connectivity. In particular, it is

$$\begin{aligned} &\forall i, j \,.\, (z^- = i \wedge z^+ = j) \Rightarrow (b^-_{i,j} = \text{water/stone}) \\ &\quad \wedge (b^+_{i,j} = \text{connected}) \wedge (\iota^+_r = \iota^-_r - 1) \\ &\quad \wedge \Big(\forall i', j',\ (b^+_{i',j'} = \text{connected}) \Rightarrow \\ &\qquad\qquad \big((b^-_{i',j'} = \text{connected}) \vee \mathcal{X} \big) \Big) \\ &\quad \wedge \Big(\forall i', j',\ (b^+_{i',j'} \neq \text{connected}) \Rightarrow (b^+_{i',j'} = b^-_{i',j'}) \Big) \\ &\quad \wedge \mathcal{Q}, \end{aligned}$$

where

$$\begin{aligned} \mathcal{X} = (b^-_{i',i} = \text{connected} \vee b^-_{i',j} = \text{connected}) \\ \wedge\, (b^-_{j',i} = \text{connected} \vee b^-_{j',j} = \text{connected}) \\ \mathcal{Q} = (\omega^+ = \omega^-) \wedge (\rho^+ = \rho^-) \wedge (\iota^+_{\setminus r} = \iota^-_{\setminus r}). \end{aligned}$$

B. Prototype Components for Box World

In this section, we describe the prototype components for the box world. They are all of the form "get k", where $k \in K$ is a color in the set of possible colors in the box world. First, we define the following abstraction variables:

- **Box**: $b_{k_1,k_2} = n$ indicates that there are n boxes with key color k_1 and lock color k_2 in the map
- **Loose key**: $\ell_k = b$, where $b \in \{\text{true}, \text{false}\}$, indicates whether there exists a loose key of color k in the map
- **Agent's key**: $\iota_k = b$, where $b \in \{\text{true}, \text{false}\}$, indicates whether the agent holds a key of color k

As in the craft environment, we use b^-, ℓ^-, ι^- and b^+, ℓ^+, ι^+ to denote the initial state and final state for a prototype components, respectively. Since the configurations of the map in the box world can only contain at most one loose key, we add cardinality constraints $\text{Card}(\ell) \leq 1$, where $\text{Card}(\cdot)$ counts the number of variables that are true.

Then, the logical formula defining the prototype component "get k" is

$$\mathcal{X} \vee \mathcal{Y},$$

where

$$\mathcal{X} = \ell_k^- \wedge \iota_k^+ \wedge (\text{Card}(l^+) = 0) \wedge (b^+ = b^-)$$
$$\mathcal{Y} = (\text{Card}(\iota^-) = 1) \wedge \iota_k^+ \wedge \neg\iota_k^- \wedge (l^+ = l^-) \wedge \Big(\forall k_1 \,.\, \iota_{k_1}^- \Rightarrow \Big((b_{k,k_1}^+ = b_{k,k_1}^- - 1) \wedge (b_{\setminus(k,k_1)}^+ = b_{\setminus(k,k_1)}^-)\Big)\Big)$$

In particular, $\mathcal{X}$ encodes the desired behavior when the agent picks up a loose key k, and $\mathcal{Y}$ encodes the desired behavior when the agent unlocks a box to get key k.