A Additional Related Work

Structured MDPs. Many works have postulated that designing algorithms for structured MDPs will lead to improvements in sample efficiency and generalization over existing algorithms for the standard MDP formulation. The two closest types of MDP families to our work are factored MDPs [3, 4, 17], relational MDPs [37], and object-oriented MDPs [9]. Factored MDPs [3, 4, 17] assume that the environment can be represented by discrete attributes, and that transitions between these attributes can be modeled as a Bayesian network. Our work differs from these in that we do not assume access to these attributes and the dependency graph is also not assumed to be known. More importantly, the focus in this work is on using attributes to factorize a task into subcomponents; and in particular being able to generalize to new, more complex tasks at test time. Our approach is also related to Relational MDPs and Object-Oriented MDPs [1, 9, 12], where states are described as a set of objects, each of which is an instantiation of canonical classes, and each instantiated object has a set of attributes. Our work is especially related to [18], where the aim is to show that by using a relational representation of an MDP, a policy from one domain can generalize to a new domain through planning. However, we discover both objects (identities) and attributes (states), and achieve generalization through factorized planning, which grows linearly (rather than polynomially) in the number of factors.

B Implementation Details

This section details the implementation design decisions for each component of HA.

B.1 Level 1: abstracting visual features into sets of entities

Dynamic SLATE (dSLATE) maps a video demonstration to a sequence of transitions over sets of entities. It consists of two main components: SLATE [32] and a transformer [28, 35] dynamics model. Because SLATE itself also uses a transformer for generating image reconstructions, dSLATE uses two transformers: one to recover the observation model E and one to recover the dynamics model P of the structured MDP. We use SLATE $\prod_k \mathcal{H}^k \times \mathcal{O} \to \prod_k \mathcal{H}^k$ to infer entities $\mathbf{h}_t$ from observation o_t and initial guess $\hat{\mathbf{h}}_t$. The entity h^k is also referred to as a *slot* in [25, 32] and is split in half as $h^k = (z^k, s^k)$. The first guess for each entity $\hat{h}_1^k$ is sampled independently and identically distributed from a unit Gaussian, whose parameters are also trained. The hyperparameters are given in Tab. 5.

SLATE preprocesses the image with a discrete variational autoencoder [29] into a grid of image features, encodes these features into a grid of tokens, infers slots from this token grid with Slot Attention [25], which also produces an attention mask `attn` over the features each slot attends to. These slots are trained using a transformer decoder [28, 35] to autoregressively reconstruct the tokens using the slots as keys/values.

We use the dynamics model $\prod_k \mathcal{S}^k \times \mathcal{A} \to \prod_k \mathcal{S}^k$ to predict a guess for $\hat{\mathbf{s}}_{t+1}$ given the action a_t and the inferred entities $\mathbf{s}_t$ from the previous time-step. This dynamics model is implemented also as a transformer decoder, taking the entity states as queries, and the entity states and action as keys/values. This enables us the dynamics model to model the future state of the slots as an equivariant function of how the action affects it and of how it interacts with other entity states.

We trained dSLATE on an offline dataset of 5000 video demonstrations of length five, with each frame transition showing one of four objects being moved to a different location. We used five slots, one more than the number of objects, following the convention used in Van Steenkiste et al. [34], Veerapaneni et al. [36].

B.2 Level 2: abstracting transitions over sets of entities into individual state transitions

HA constructs a graph of state transitions from a buffer of transitions over entity sets produced by dSLATE, as described in Alg. 1. Each transition records the identity z, state s, and `attn` of each entity of the entity sets $\mathbf{h}_t$ and $\mathbf{h}_{t+1}$. We also record the *pre-condition* of the transition, which we explain further in Appdx. B.3. Both s or `attn` can serve as the representation of the state. We found that we obtained better clusterings when we used `attn` as the state for the *block-** tasks and s as the state for the *robogym-rearrange* task. We also empirically found that certain choices of distance metric used for K-means `clustering` and `binding` (implemented as nearest-neighbors) depended

Number of epochs		200
Episodes per epoch		5K
Episode length		5
Batch size		32
Peak LR		(see caption)
LR warmup steps		30000
Dropout		0.1
Discrete VAE	Vocabulary Size	4096
	Temp. Cooldown	1.0 to 0.1
	Temp. Cooldown Steps	30000
	LR (no warmup)	0.0003
	Image Size	(see caption)
	Image Tokens	Image Size / 4
transformer decoder	Layers	4
	Heads	4
	Hidden Dim.	192
Slot attention	Slots	5
	Iterations	3
	Slot Heads	1
	Slot Dim. (h^k)	192
	Identity Dim. (z^k)	96
	State Dim. (s^k)	96
transformer dynamics	Layers	4
	Heads	4
	Hidden Dim.	96

Table 5: **Hyperparameters for training dSLATE** These hyperparameters are almost identical to those found in Singh et al. [32, Fig. 7], but because dSLATE operates on video demonstrations rather than static images, we changed some hyperparameters to save memory cost. We changed the batch size from 50 to 32, the number of transformer layers and heads from 8 to 4, the number of slot attention iterations from 7 to 3 without observing a significant change in performance. We used a peak learning rate of 0.0002 and an image size of 64 for *-*rearrange*. We used a peak learning rate of 0.0003 and an image size of 96 for *block-stacking*.

on which choice of state representation we used, and this is summarized in Table 6. The K-means implmentation is adapted from `https://github.com/overshiki/kmeans_pytorch`. We found that increasing the number of slot attention iterations improved the entities representations especially when generalizing to more numbers of objects, so even though we dSLATE trained with slot attention three iterations, for inferring the slots from the buffer we used seven iterations. Lastly, we found that the number of clusters used to for K-Means is the most important hyperparameter for creating a graph that reflected the state transitions. We swept over 16 to 50 clusters and report the optimal number of clusters we found in Table 7.

State representation	`attn`	s
`isolate` distance metric $d(\cdot, \cdot)$	cosine	cosine
`cluster` distance metric	IoU	squared Euclidean
`bind` distance metric	cosine	squared Euclidean

Table 6: **Hyperparameters for constructing the transition graph with HA**

	block-rearrange	*robogym-rearrange*	*block-stacking*
number of clusters	30	45	47

Table 7: **Number of clusters used for constructing the nodes of the transition graph.**

B.3 Action selection

Using dSLATE and the transition graph from HA, HA returns which action to execute in the environment given a goal and current observation, as described in Alg. 2. It first infers goal constraints $\mathbf{h}_g$ and current entities $\mathbf{h}_t$ from the goal observation o_g and current observation o_t. It then uses the `align` procedure to align the indices of the entities in $\mathbf{h}_g$ and $\mathbf{h}_t$ and uses the `select-constraint` to choose the index k of the entity to affect. It binds h_t^k and h_g^k to the graph and returns the action associated with the edge between their respective nodes. Because HA is a non-parameteric method, it could be the case the graph does not contain such an edge. In this case, we sample a random action in the environment, but future work will replace this step with a more sophisticated method.

To implement `align` we use the `scipy.optimize.linear_sum_assignment` implementation of the Hungarian algorithm, with Euclidean distances between the z^k's as the matching cost.

In `select-constraint`, we are given the set of current entities $\mathbf{h}_t$ whose indices are aligned with the goal constraints $\mathbf{h}_g$ and returns the index k of the goal constraint to satisfy next. By HA' construction, the edge between the nodes that h_t^k and h_g^k are bound to is the state transition that would be executed if the action associated to the edge were taken in the environment. The `select-constraint` procedure consists of three steps: (1) filtering possible transitions from impossible transitions (2) ranking transitions (3) sampling a transition.

Filtering The filtering step implements HA' model of possibility and impossibility. In the filtering step, we consider, for each k, the transition between the nodes that h_t^k and h_g^k are bound to and mark the transition as possible or impossible. It then returns the indices k over $\mathbf{h}_t$ and $\mathbf{h}_g$ whose associated transition from h_t^k and h_g^k is possible.

According to HA, a state transition between node $[i]$ and node $[j]$ is *possible* if its preconditions are met and there exists an edge for that state transition in the graph, and *impossible* otherwise. An intuitive example of an impossible transition is to place a block midair at some intended height, but this transition becomes possible if prior to the transition there already exists a stack of blocks that would support the block if the block were to be placed at that intended height. The existence of this supporting stack is thus the *precondition* for the transition to occur, rendering the stacked block *dependent* on the blocks supporting it.

When there are no dependencies among the entities, as in the **-rearrange* tasks where any object can be moved to any open location without considering where other objects are, any transition present in the graph is possible. When there are dependencies among the entities, as in *block-stacking*, we take the precondition of the transition into account. Although a precondition of a transition from node $[i]$ to node $[j]$ could be a function of both the source node $[i]$ and destination node $[j]$, for simplicity in this paper we consider preconditions as only a function of the destination node $[j]$, which rules out the possibility of placing a block in midair like the above example.

Because a precondition in general is a set of constraints that need to be satisfied for the transition to be possible, we represent the precondition of a transition into node $[j]$ (denoting the state $s_*^{[j]}$) as the set of context states $s_*^{[j']}$ that are always present whenever the state $s_*^{[j]}$ is present. Concretely, a block at height 3 at location x is always accompanied by the presence of some block at height 2 and some block at height 1, both at location x; the states denoting (location x, height 2) and (location x, height 1) are the context states of the state (location x, height 3) and therefore serves as its precondition. We thus implement the precondition of a transition into node $[j]$ by recording the indices j' of the nodes of states that are always present when node $[j]$ is a destination node. To test whether a precondition is satisfied for a given scene, we check if all nodes in the precondition set have a corresponding concrete entity that can be bound to it.

Ranking The filtering step removes the indices from the entities $\mathbf{h}_t$ and goal constraints $\mathbf{h}_g$ whose transitions are impossible, yielding a possibly smaller set of entities $\tilde{\mathbf{h}}_t$ and constraints $\tilde{\mathbf{h}}_g$. That is, if $|\mathbf{h}_t| = |\mathbf{h}_g| = K$, then $|\tilde{\mathbf{h}}_t| = |\tilde{\mathbf{h}}_g| = \tilde{K} \leq K$.

The goal of the ranking step is to compute a ranking among the indices of $\tilde{\mathbf{h}}_t$ and $\tilde{\mathbf{h}}_g$ for choosing which index k to actually select to affect with an action. Intuitively, we should rank indices k

according to how different s_t^k and s_g^k are because a difference indicates that the constraint h_g^k is not satisfied. We reuse the distance metric $d(\cdot, \cdot)$ used for `isolate` to implement this ranking.

Sampling The goal of the sampling step is to select a $k \in \{1, ..., \tilde{K}\}$ whose associated entity we will affect with an action, given our ranking. One way to do this is to simply choose k as $k = \arg\max_{k' \in \{1,...,\tilde{K}\}} d(s_t^{k'}, s_{t+1}^{k'})$ as in `isolate`, but we empirically found that sampling k from a Categorical distribution whose pre-normalized probabilities are given by $d(s_t^{k'}, s_{t+1}^{k'})$ resulted in better task performance so we used this stochastic sampling approach. One explanation for why using the argmax may be worse is that it relies on the distance metric $d(\cdot, \cdot)$, and the state representation s, to be such that the distance metric flawlessly assigns high value to entities k that need to be moved and low value to entities k that do not need to be moved. But because the state space $\mathcal{S}$ is learned through the dSLATE training process without explicit supervision on the geometry of the space, a pair of points that should be farther apart than another set of points may not be accurately reflected by using a fixed distance metric $d(\cdot, \cdot)$. Future work will investigate imposing explicit supervision on the geometry of $\mathcal{S}$.

C Baseline Implementation Details

Random (Rand) The random policy takes actions using `env.action_space.sample()`.

Behavior cloning (BC) This approach trains a policy to output the actions directly taken in the provided dataset. We use an MSE loss to train the policy to imitate the actions.

Implicit Q-learning (IQL) IQL is a simple, offline RL approach that uses temporal difference (TD) learning with the dataset actions and trains a behavior policy value function. To produce an optimal value function, IQL estimates the maximum of the Q-function using expectile regression with an asymmetric MSE using the following objectives:

$$L_V(\psi) = \mathbb{E}_{(s,a)\sim\mathcal{D}}[L_2^\tau(Q_{\hat{\theta}}(s,a) - V_\psi(s))] \text{ where } L_2^\tau(u) = |\tau - \mathbb{1}(u < 0)|u^2 \quad (1)$$

$$L_Q(\theta) = \mathbb{E}_{(s,a,s')\sim\mathcal{D}}[(r(s,a) + \gamma V_\psi(s') - Q_\theta(s,a))^2] \quad (2)$$

$$L_\pi(\phi) = \mathbb{E}_{(s,a)\sim\mathcal{D}}[\exp(\beta(Q_{\hat{\theta}}(s,a) - V_\psi(s))) \log \pi_\phi(a|s)]. \quad (3)$$

The $V(s)$ estimates are used for TD-backups and the optimal policy is extracted with advantage-weighted behavioral cloning.

Model predictive control (MPC) This approach uses model predictive control with the cross entropy method (CEM) to select actions, using the transformer dynamics model of dSLATE to perform rollouts in latent space. This is similar to the approached used in OP3 [36], except that we use more recently proposed architectural components (slot attention [25] instead of IODINE [15], a transformer instead of a graph network [2, 7, 34]) so our MPC results are not directly comparable to that of OP3. We use the same dSLATE checkpoint that was used for HA.

We implement this MPC baseline using the `mbrl-lib` library [27] with 10 CEM iterations, an elite ratio of 0.05, and a population size of 250 which was the best configuration we found that fit within a wall clock budget of two days for 8 objects and 100 test episodes. We swept over CEM iterations of $[5, 10, 20]$, elite ratio of $[0.05, 0.1, 0.2]$, and population sizes of $[250, 500, 1000]$, and found that the elite ratio was the most important hyperparameter.

The cost function is computed by first aligning the predicted slots $\mathbf{h}_T$ and goal constraints $\mathbf{h}_g$ using the same `align` procedure in B.3, and then adding up the squared Euclidean distance between slots as $cost = \sum_k (h_T^k - h_g^k)^2$.

Monolithic graph search (MGS) This approach is an ablation to HA that does not construct a graph over state transitions of individual entities but instead constructs a graph over state transition over entity sets, i.e. each transition is $(\mathbf{s}, a, \mathbf{s}')$ rather than $(s^k, a, s^{k\prime})$. As with MPC, we use the same dSLATE checkpoint that was used for HA.

The purpose of this ablation is to elucidate the benefit of factorizing the transition graph over entities rather than entity sets. Because nodes in this transition graph represent a set of entity states rather

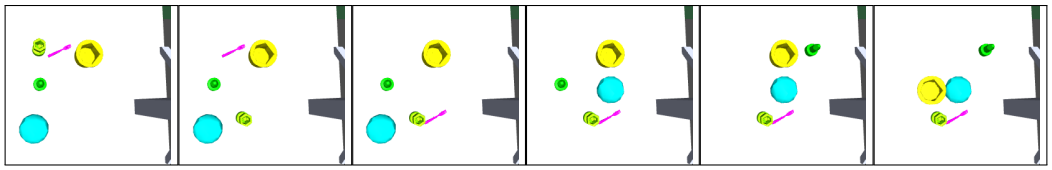


Figure 5: An example of solving a task in the robogym rearrange environment used in this paper.

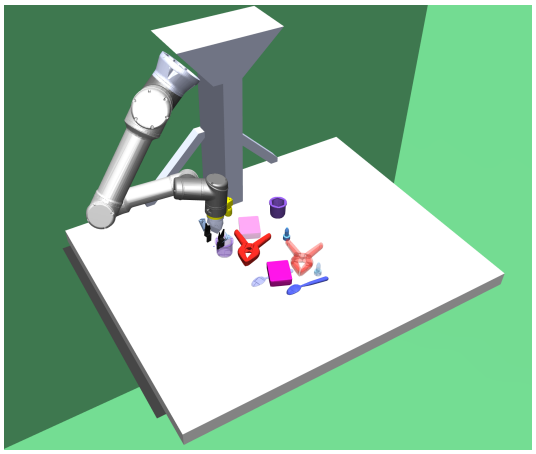


Figure 6: The original Robogym rearrange setup

than individual entity states, we use Dijkstra's algorithm, as in [10, 38, 39] to plan a unbroken path from the node the initial observation is bound to to the node a goal observation is bound to. For each time-step, we plan a path along the nodes using Dijkstra's algorithm, then return the action associated with the first edge along that path. Like HA, MGS is a non-parametric model, which means that for a set of entities to be bound to a node in the graph, that node must contain the exact set of entity states corresponding to the states of the entities. If we do not successfully bind to the graph, or if we do not find a path between the current node and the goal node, we sample a random action as HA does.

D Environment Details

Block-rearrange is our simplest environment, and *robogym-rearrange* and *block-stack* each add complexity along different axes. States and identities correspond to object locations and appearance respectively. In *block-rearrange* (Fig. 2a), all objects are the same size, shape, and orientation. $\mathcal{S}$ covers 16 locations in a grid. $\mathcal{Z}$ is the continuous space of red-green-blue values from 0 to 1. *robogym-rearrange* (fig. 2b) adapts the rearrange environment from OpenAI [26] and removes the assumption from *block-rearrange* that all objects are the same size, shape, orientation, and the assumption of predefined locations. *block-stack* (fig. 2c) adds preconditions and postconditions on whether objects can be moved: blocks can only be picked if from the top of a stack, and blocks can only be placed at a given height if there is an object beneath to support it (otherwise it falls). *Block-rearrange* and *Block-stack* are implemented in PyBullet [8] while *Robogym-rearrange* is implemented in Mujoco [33].

Environments In *block-rearrange*, all objects are the same size, shape, and orientation and can exist in any one of 16 locations in a grid. Colors are sampled in a continuous space of red-green-blue values in $[0, 1]$.

The *robogym-rearrange* environment (see figures 5 and 6) is adapted from the rearrange environment in OpenAI's Robogym simulation framework [26] and removes the assumption from *block-rearrange* that all objects are the same size, shape, and orientation and the assumption of predefined locations. Furthermore, due to 3D perspective, the objects can look slightly different in different locations. The objects are uniformly sampled from a set of 94 meshes consisting of the YCB object set [5] and a set of basic geometric shapes, with colors sampled from a set of 13. The camera angle is a bird's eye view over the table, and the size of each object is normalized by its longest dimension, so tall

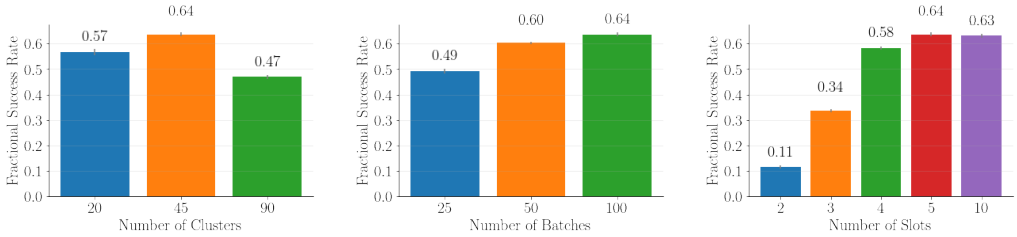


Figure 7: The performance of our method as the number of initialized clusters and batches from the training set used to construct the graph, and the number of slots are varied.

thin objects appear smaller. The objects' target positions are randomly sampled such that they don't overlap with each other or any of the initial positions, and the target orientation is set to be unchanged. Due to the continuous nature of this environment, we define a match threshold of at most 0.05 for both the initial pick position and the goal placement (the table dimensions are 0.6 by 0.8).

The *block-stack* environment adds preconditions and postconditions on whether objects can be moved: blocks can only be picked if they are at the top of a stack, and blocks can only be placed at a given height if there is already an object beneath to support it (otherwise it falls).

Sensorimotor interface Each observation is a tuple of an initial image displaying the current observation and a goal image displaying constraints to be satisfied – the goal locations of the objects. Each action is a tuple $(w, \Delta w)$, where w is a three-dimensional Cartesian coordinate (x, y, z) in the environment arena.

For the $*$-*rearrange* tasks, objects are initialized at random non-overlapping locations that also do not overlap with their goal locations. For these tasks the z (height) coordinate is always fixed. For the *block-stack* task, the goal locations are generated by randomly picking objects from the tops of stacks and placing them on other stacks. For this task the y (depth) coordinate is always fixed.

An object is picked if w is within a certain threshold of its location. For *block-*$*$ where object locations are fixed points in a grid the object is snapped to the nearest grid location to $w + \Delta w$. Constraints are considered satisfied if objects are placed within a certain threshold of their target location.

E Additional Results

This section presents additional results and analyses of HA. We first analyze the sensitivity of task performance to several hyperparameters used in HA from creating the graph: the number of clusters, number of buffer size, and the number of slots used in slot attention. We next tested whether giving our model-based baselines more computation time compared to HA would improve their performance to be comparable to HA's.

E.1 Analysis of key hyperparameters

In this section, we analyze the sensitivity of our method to various hyperparameters, evaluated in the robogym environment with four objects in the complete goal specification. As Fig. 7 shows, performance depends on the number of initialized clusters and the number of batches from the training set used to construct the graph. With too few clusters, the clusters are too coarse-grained to differentiate objects in significantly different positions, while with too many the performance deteriorates as the data is needlessly split into duplicate clusters. Performance improves with more data, as the graph has better coverage. Although our model performs worse when there are insufficient slots to represent all objects present in the environment plus one empty slot, performance is barely impacted by having double the number of necessary slots. Our method can thus still work in environments with an unknown but upper-bounded number of objects.

E.2 More challenging evaluation settings

We analyzed HA in more challenging settings that reflect the noisy nature of real-world robotics. As Fig. 8 (left) shows, HA is robust to the addition of Gaussian noise to the action at every time step, up until the noise variance is comparable to the maximum distance for successful picking and goal placements. The performance also remains high given significantly fewer steps (Fig. 8, right).

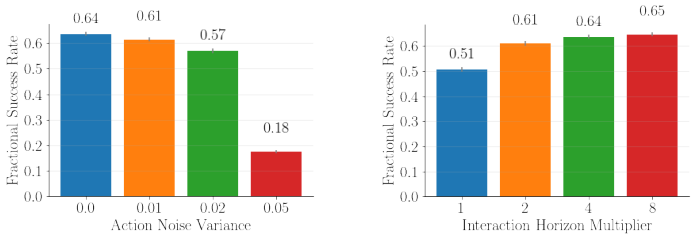
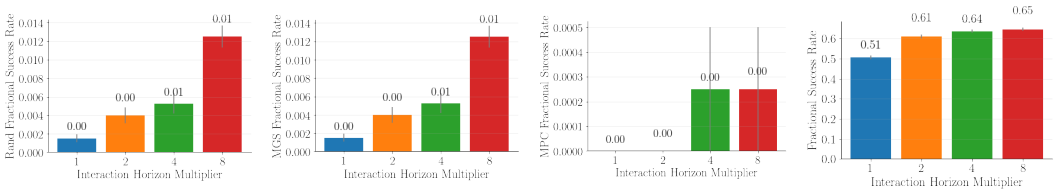


Figure 8: The performance of HA on *robogym-rearrange* as we vary the amount of noise added to the actions, and the interaction horizon (as a multiple of the minimum steps needed to complete the task).

E.3 More computation time for model-based baselines

We tested whether doubling the computation time for the model-based baselines would improve their performance to be comparable to HA's. For the results in the main paper, we capped the length of the episode as 4x the minimum number of actions required to solve the task. In Fig. 9, we vary this interaction horizon multiplier from 1x to 8x. HA degrades less with shorter interaction horizons compared to the baselines. We find that MGS performs similar to the random baseline. Since MGS takes a random action if it cannot bind the given entity set to its graph, this result suggests that the space of subsets of entities is so combinatorially large that MGS does not successfully bind to the graph most of the time. We verified that this is the case by inspecting when MGS takes random actions. MPC performs the worst out of all the methods, performing worse than random. We tested that the cost function described in C ranks latents that match the goal constraint with lower cost than randomly sampled latents, which suggests that the main source of error is due to the inaccuracy in the prediction rollouts. This can be expected, as learned models suffer from compounding errors when rolled out [20] and prior methods that use MPC for object-centric methods only roll out for very short horizons [36].



(a) Rand (b) MGS (c) MPC (d) HA

Figure 9: **Varying interaction horizon.** The performance of the MGS (b) and MPC (c) baselines compared to HA (d, reproduced from Fig. 8) and the random baseline (a) on *robogym-rearrange* as we vary the interaction horizon (as a multiple of the minimum steps needed to complete the task). *Note that the scale of the y-axis is not the same.* While a longer horizon improves performance, HA still achieves at least 50x better accuracy with an interaction horizon multiplier of 1 than the performance obtained by increasing the interaction horizon multiplier for the model-based baselines to 8.