Learning to Compose Hierarchical Object-Centric Controllers for Robotic Manipulation

Mohit Sharma†1
Robotics Institute
Carnegie Mellon University

Jacky Liang†1
Robotics Institute
Carnegie Mellon University

Jialiang Zhao1
Robotics Institute
Carnegie Mellon University

Alex LaGrassa1
Robotics Institute
Carnegie Mellon University

Oliver Kroemer1
Robotics Institute
Carnegie Mellon University

1{mohitsharma, jackyliang, alanjz, alagrass, okroemer}@cmu.edu

Abstract: Manipulation tasks can often be decomposed into multiple subtasks performed in parallel, *e.g.*, sliding an object to a goal pose while maintaining contact with a table. Individual subtasks can be achieved by task-axis controllers defined relative to the objects being manipulated, and a set of object-centric controllers can be combined in an hierarchy. In prior works, such combinations are defined manually or learned from demonstrations. By contrast, we propose using reinforcement learning to dynamically compose hierarchical object-centric controllers for manipulation tasks. Experiments in both simulation and real world show how the proposed approach leads to improved sample efficiency, zero-shot generalization to novel test environments, and simulation-to-reality transfer without fine-tuning.

1 Introduction

Manipulation tasks are inherently object-centric and often require a robot to perform multiple subtasks in parallel, such as pressing on a sponge while wiping across a surface, balancing a saucer while serving tea, or maintaining alignment of a screwdriver while unscrewing a screw. The individual subtasks need to be performed in parallel to accomplish the overall task. As the above examples illustrate, subtasks usually correspond to goals and constraints associated to objects in the robot's environment. Thus, manipulation skills are often defined as 3D motions, which are implemented as simple position or force controllers, of the end effector in object-centric coordinate frames.

One drawback of such an approach is that it results in monolithic controllers for each task, *i.e.* controllers which act specifically with respect to some fixed coordinate frame. In addition, for many tasks it is not always necessary to control all axes of a given object-centric coordinate frame. For instance, for the wiping task in Figure 1, the sponge needs to use the table surface normal to make contact with the surface, while it is free to move with respect to any other object (wall, corners, dirt) on the surface. Based on this insight, we adopt a modular approach by defining task-axis controllers for each potential subtask. Importantly, the controllers are associated with object-centric axes, such as the normal of a surface or the direction from the end-effector to an object.

We focus on learning an hierarchy of such object-centric task-axis controllers, or **object-axis controllers** (Figure 1). This hierarchy is especially important since many tasks require performing multiple subtasks in parallel. Previous works use pre-defined sets of task frames attached to objects or the robot, and they often learn a fixed task-frame hierarchy from human demonstrations. Instead, we use Reinforcement Learning (RL) to learn a policy that outputs an ordered list of controllers, which are then composed to be executed on the robot. To ensure different object-axis controllers

† Equal Contribution

4th Conference on Robot Learning (CoRL 2020), Cambridge MA, USA.

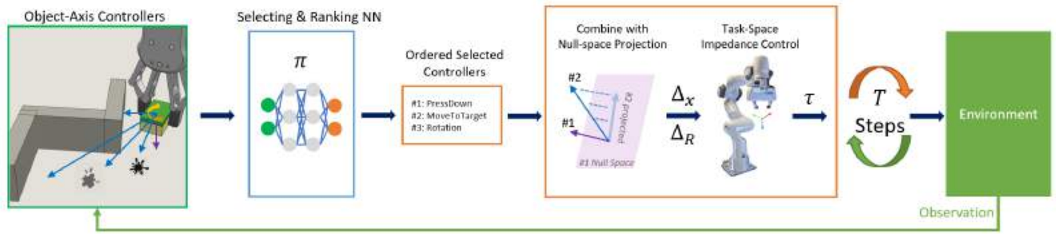


Figure 1: Controller Selection and Composition Pipeline. Given current observations and list of low-level controllers, an RL policy chooses an ordered list of controllers to use. These controllers are composed via nullspace projection, where the controls of lower-priority controllers are projected onto the nullspace of higher-priority ones. The combined control signals are used to actuate a robot via task-space impedance control. The controller combination runs for T time steps before the RL policy is queried again.

do not interfere with each other, we compose controllers via nullspace projections [1], where the control signals of lower-priority controllers are projected onto the nullspace of higher priority ones.

In addition to modularity, our approach provides several other benefits. First, the object-axis controllers are not task specific, so they can be reused across multiple tasks. Second, composing controllers across multiple different objects makes the learned policies invariant to certain object properties *e.g.*, a controller that reaches toward an object is invariant to object size. Such invariances are useful for generalizing learned policies beyond the set of objects the policies are trained on. Finally, the use of a structured action space introduces meaningful inductive biases by ensuring robot actions are performed both in relation and with respect to objects in the scene. We successfully evaluated our approach on four different manipulation tasks, including two 2D tasks of fitting and pushing a block and two real robot tasks of screwing and door-opening. Experiments show that the proposed approach leads to improved sample efficiency, zero-shot generalization to novel environment configurations, and simulation-to-reality transfer without further fine-tuning. See videos and supplementary materials at `https://sites.google.com/view/compositional-object-control/`.

2 Related Works

Task Frames: Our use of task-axes is related to the notion of 6D task frames [2, 3, 4]. One of the first works to formalize task frames is [4]. There, the authors referred to different task-axes as compliant or non-compliant based on the type of desired motion along each axis. The authors of [5] proposed hybrid force-position control, which selects different axes of the constraint frame for either position or force control. Simultaneously, the authors of [2, 3] proposed task frames to define robotic manipulation primitives; they noted that the geometric level of task frames can serve as a good middle ground between symbolic actions and the motor control input. Since then, task frames in the form of task spaces have been used extensively in robotics [6]. Prior works treat task-frames as fixed coordinate frames which are either attached to objects of interest or generated from constraints in the environment. By contrast, our approach is more modular and dynamic, as it enables an RL policy to combine task-axes across different objects and dynamically synthesize task-frames.

Task Frame Selection: Although the use of task frames and spaces is widespread in robotics [7, 8, 9, 10, 11, 12, 13], only a few works have explored using learning to select which task frames are appropriate for the given task [7, 10, 11, 14, 15]. However, most of these works use imitation learning *i.e.*, they learn task frame selection from human demonstrations [7, 10, 11, 14]. The criterion for task-frame selection is typically manually defined using properties such as inter-trial variance or convergence behavior of demonstrations. In our work, we set task-axes selection as the action space for an RL agent, so we do not require demonstrations. Moreover, the RL agent chooses a hierarchy of task-axis controllers, which are composed together for execution.

Hierarchical Controllers: Combining multiple task-axis controllers is related to works in hierarchical control. Hierarchical control is often used in robots with redundant degrees of freedom or bi-manual robot setups where multiple tasks or objectives can be executed in parallel [16, 17, 18, 19]. To combine different controllers, these works project the control signals of lower-priority controllers onto the nullspace of higher-priority controllers. However, most of these works assume a fixed priority order for the tasks/objectives being considered, while some recent works [19] learn the priorities from human demonstrations. Similar to these works, our approach also uses nullspace projections

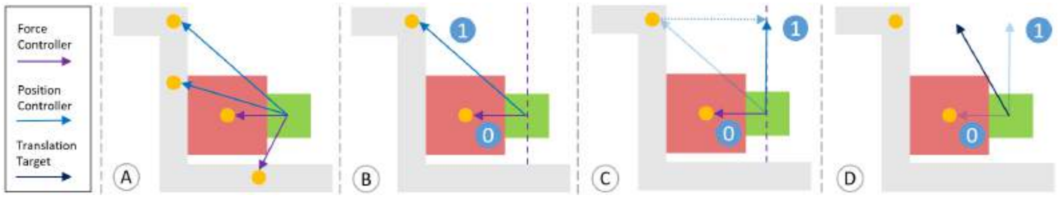


Figure 2: Force-Position Controller Composition. Here, the agent controls the green block to push the red block up along the vertical gray wall. A) The agent is given 4 controllers to choose from, each corresponding to points of interests in the scene. B) The agent chooses 2 controllers, with the force controller into the red block at the higher priority (0), and position controller toward the wall corner at the lower priority (1). C) The error of the lower-priority position controller is projected onto the null space of the higher-priority force controller (purple dashed line). D) The projected errors are combined to form the desired position target.

to combine multiple task-axis controllers together. However, instead of using a fixed priority order, our method learns to prioritize controllers by directly interacting with the environment.

Reinforcement Learning: Finally, our approach is related to works on structured action spaces for reinforcement learning (RL) for contact-rich manipulation tasks. Recent works have studied how the choice of action spaces affect robot learning performance [20, 21, 22]. However, these methods focus only on the final controller output, *i.e.*, comparing fixed with variable impedance control [20, 21] or with hybrid-force position control [22] in joint and task-spaces. Our work provides additional structure to the action space via composing hierarchical object-centric controllers.

Hierarchical RL: Composing task-axes controllers for performing tasks is also related with hierarchical RL (HRL) [23, 24, 25]. HRL uses the notion of options, which are temporally-extended actions, and learns to combine them to accomplish a given task. There has been a large body of work which aims to extract the underlying options [26, 27, 28], using techniques such as bottleneck states [26], policy sketches [29], or expert demonstrations [30, 31, 32]. Similarly, there have also been works that use predefined option policies and compose them to learn a "meta-policy" [33]. However, these option policies are defined specifically with respect to the underlying task, and hence it is not clear how reusable these policies are. By contrast, our proposed task-axes controllers are reusable across multiple different manipulation tasks. This is desirable for efficient learning of new manipulation tasks[34]. Additionally, task-axes controllers are different than options since they can be composed both hierarchically and temporally.

3 Learning Hierarchical Compositions of Object-Centric Controllers

We propose training an RL policy to perform manipulation tasks by using a structured action space consisting of hierarchical compositions of object-centric controllers. Each object in the scene is associated with a fixed set of task-axes, positioned either at object centers or other object key points. For each axis, we define a set of controllers that perform force, position, and rotation controls. This gives a set of pre-defined object-centric task-axis controllers, or object-axis controllers, which define our structured action space. With this action space, instead of directly commanding the end-effector, the RL policy selects multiple object-axis controllers in a prioritized order, which are composed together using null-space projections. Figure 1 shows an overview of the overall proposed approach.

In the next subsections, we first define the different types of object-centric low-level controllers we use, including how their object-centric axes are defined. We then discuss how to combine different object-axis controllers together using null-space projections. Finally, we discuss different RL approaches for learning the high-level policy that selects multiple controllers.

3.1 Controller Types

In this work, we use three different types of controllers: position, force, and rotation. These controllers are object-centric, *i.e.* their control targets and axes correspond to objects in the scene. For example, position controllers could be attractors that lead the end-effector (EE) close to an object of interest, force controllers could be applying forces perpendicular to object surfaces, and rotation controllers could be aligning an axis of the EE with an axis of the object. Currently, these controllers are manually specified (see details in Section 4), but they could also be autonomously inferred from visual observations of objects in the environment. Figure 2 illustrates force and position controllers and their composition, and Figure 3 shows the rotation controllers.

Let $x_c \in \mathbb{R}^3$, $R_c \in SO(3)$, and $f_c \in \mathbb{R}^3$ respectively denote the current end-effector position, orientation, and forces expressed in the robot's base frame.

Position and Force Controllers: The position controller consists of a target position x_d and an axis u along which the controller will move the robot's end-effector toward the target. u can be a fixed direction, like the normal direction of a surface, or it can be adapted with respect to x_c: $u = \frac{x_d - x_c}{\|x_d - x_c\|_2}$. Let $\mathcal{P}(u) = uu^\top$ be the projection matrix for the given axis. Then, the translation error a position controller produces is defined as $\delta_x(x_d, u, x_c) = \mathcal{P}(u)(x_d - x_c)$. The force controller is similar to the position controller, *i.e.* given a force target f_d and an axes-direction u, the force error the controller produces is $\delta_f(f_d, u, f_c) = \mathcal{P}(u)(f_d - f_c)$.

Rotation Controller: The rotation controller attempts to align one axis $R_c u$ of R_c with a target axis r_d, where u is a unit vector that performs axis-selection. For example,to align the X-axis of the end-effector frame to align with r_d, then $u = [1, 0, 0]^\top$. The rotation controller produces a delta rotation target in the end-effector frame, which we compute via the angle-axis representation: $\delta_R(r_d, u, R_c) = \cos^{-1}((R_c u)^\top r_d)((R_c u) \times r_d)$

Null Controllers: The high-level policy also has the option to choose a null controller, which would give 0 errors for both δ_x and δ_R. While other controllers can be chosen at most 1 time, the null controllers can be chosen multiple times, giving the high-level policy more flexibility.

3.2 Controller Composition

Force-Position Composition: The RL policy selects at most 3 force and position controllers to compose. Only 3 of force and position controllers can execute concurrently, because there are only 3 position dimensions. The RL policy outputs a priority order for these controllers. Let the indices $[0, 1, 2]$ denote the 3 controllers in decreasing priority, so 0 is the highest, and 2 the lowest. The final position target is computed by projecting the lower-priority targets onto the nullspaces of the higher-priority controllers, then summing them. Let $\mathcal{N}(U) = I - U^\dagger U$ be a nullspace projection matrix with respect to rows of U, where † denotes the pseudoinverse. Let K_x be the position controller gain and K_f the force gain:

$$\Delta_x^0 = K_x \delta_x(x_d^0, u^0, x_c) \tag{1}$$
$$\Delta_x^1 = K_x \mathcal{N}([u^0]) \delta_x(x_d^1, u^1, x_c) \tag{2}$$
$$\Delta_x^2 = K_x \mathcal{N}([u^0, u^1]) \delta_x(x_d^2, u^2, x_c) \tag{3}$$
$$\Delta_x = \sum_{i=0}^{2} \Delta_x^i \tag{4}$$

where $[\ldots]$ represents a concatenation operator, *i.e.* concatenation of vectors into a matrix, *e.g.*, $[u^0, u^1] \in \mathbb{R}^{2\times 3}$. Although the above expressions are written with all 3 controllers as position controllers, in our implementation we combine multiple position and force controllers together. If force controllers are used, for the corresponding controller, swap δ_x with δ_f, x_d with f_d, x_c with f_c, and K_x with K_f. Figure 2 illustrates the force-position controller composition.

Rotation Composition: The RL policy selects at most two rotation controllers to compose. This is because when the highest priority controller fixes one axis of a rotation frame, there is only one degree of freedom left, which is a rotation in the 2D nullspace of the fixed axis. Similar to force-position controller compositions, we project the errors of lower-priority controllers onto the nullspace of higher-priority controllers:

$$\Delta_R^0 = K_R \delta_R(r_d^0, u^0, R_c) \tag{5}$$
$$\Delta_R^1 = K_R \delta_R(\mathcal{N}([R_c u^0]) r_d^1, u^1, \mathcal{N}([R_c u^0]) R_c) \tag{6}$$
$$\Delta_R = \Delta_R^1 \circ \Delta_R^0 \tag{7}$$

where $\circ$ denotes composing rotations, and K_R denotes a rotation error gain. This procedure ensures the higher-priority rotation controller always reaches its goal, and the trajectory of that axis is not affected by the lower-priority controller (see Figure 3 for an illustration).

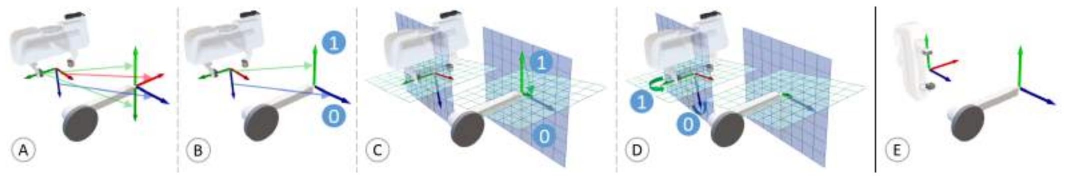


Figure 3: Rotation Controller Composition. Here, the agent rotates the Franka robot's gripper from the initial pose (A) to the final pose (E), so the gripper aligns with a door handle. A) The agent is given 4 rotation controllers to choose from, aligning various axes of the gripper with different target axes of the handle. B) Two controllers are chosen with the higher-priority labeled as (0) and the lower-priority as (1). C) Both the current and target axes of the lower-priority controller (green arrows) are projected down to the null-space (green planes) of the current axis of the higher-priority controller (gripper's blue axis). D) The desired rotation target is formed by combining the higher-priority rotation in the blue plane with the projected lower-priority rotation in the green plane. Note that the lower-priority rotation does not interfere with the higher-priority rotation.

Controlling the Robot: We use task-space impedance control to convert translation and rotation targets to configuration-space targets via Jacobian transpose, and we actuate the robot via joint torques. We first concatenate the translation target Δ_x with the axis-angle representation of Δ_R to form the final 6D delta end-effector target Δ. Then, the robot joint-torque commands are computed as $\tau = J^\top(K_S\Delta + K_D\dot{\Delta})$, where K_S and K_D are diagonal stiffness and damping matrices, and J is the analytic Jacobian. Terms for compensating gravity and Coriolis forces are omitted for brevity. In practice, we cap the magnitude of Δ to limit maximum control effort, and we add an integral term to the force controllers for better convergence. Once a set of controllers are selected, their combination runs for T timesteps before the RL policy is queried again for a new set of controllers.

3.3 RL with Object-Axis Controllers

We use RL to learn a policy that composes object-axis controllers to perform the underlying task. The policy outputs an ordered list of controllers, which are composed together to output the final control signal to move the robot. The combination of controllers is run for a fixed T timesteps, before the RL policy is queried again. Note that the controllers do not have to converge before the RL policy switches to the next combination. We next discuss multiple ways in which the RL policy can output the ordered list of controllers.

Discrete Combinatorial Actions: Let N be the total number of available controllers, and N_c be the number of controllers that can be executed simultaneously. One simple way to output an ordered list of N_c controllers is to use a discrete action space, where the policy selects an action from all available controller permutations. Such an action space grows combinatorially ($\mathcal{O}(N^{N_c})$), and is not scalable for environments with a large number of controllers.

Continuous Priority Scores: A continuous space alternative is to allow the policy to output a priority score in $[0, 1]$ for all controllers. These priority scores are then used to order the controllers, where the N_c controllers with highest priorities are executed at each step. Although the dimension of this action space grows linearly with the number of controllers, it can often lead to sub-optimal performance since the agent now needs to explore a much larger action space than before.

Expanded-MDP: To avoid the sub-optimal performance of the above methods, we propose an expanded-MDP formulation that still uses a discrete action space while avoiding combinatorial expansion. Here, we expand each environment-execution step of the MDP into N_c intermediate controller-selection steps, with the original environment-execution step occurring after the N_c'th intermediate step. At each intermediate step, the policy selects one controller from the N choices. Once N_c controllers are selected, the robot takes an actual environment step. The reward function is modified such that 0 rewards are given for the controller-selection steps before the N_c'th step. Similar MDP transformations have been suggested previously to solve continuous action MDPs using discrete action space RL algorithms [35, 36].

To use the Expanded-MDP formulation, at each controller-selection step the policy needs to know its previous controller selections. One approach is representing each controller with 1-hot encoding and appending the 1-hot encodings of previously selected controllers to the observations. This expands the observation space by $N \times (N_c - 1)$ dimensions, and we refer to this representation as **multi-1-hot**. However, in many cases it might not be necessary to know the order of the previous controllers being selected, *i.e.*, it is sufficient to know which controllers have been selected previously but not

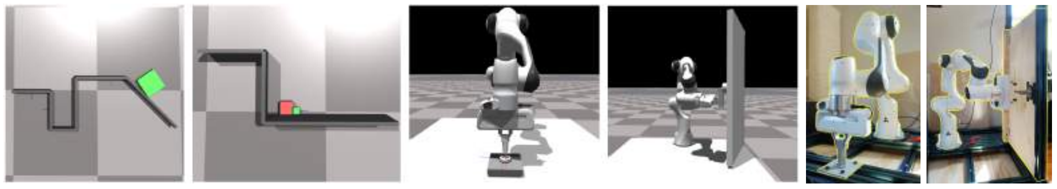


Figure 4: Experiment Tasks. From left to right: Block Fit, Block Push, Franka Hex-Screw, Franka Door-Opening tasks implemented in simulation, and Franka tasks in the real world.

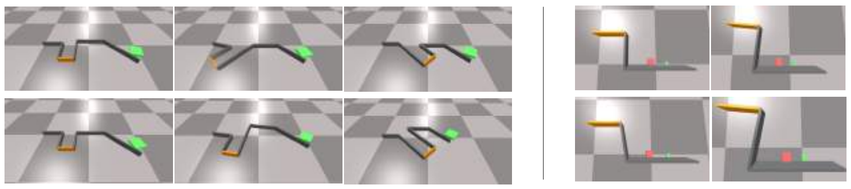


Figure 5: Example environment configurations for Block Push (left) and Block Fit (right) environments. Top row shows *some* examples of train configurations, and the bottom row shows *some* examples of test configurations. The orange wall shows the goal wall to reach.

their order. So, for the second representation, we merge the one-hot encodings of multiple previous controllers into one binary vector. This only increases the observation space by N dimensions, and can lead to faster learning. We refer to this representation as **single-1-hot**

4 Experiment Tasks and Setup

With our experiments we aim to evaluate 1) How useful are the proposed object-axis controllers for task learning, 2) How important is controller composition for task learning, and 3) How well does our proposed approach generalize to the different test configurations.

Figure 4 visualizes the tasks used to evaluate our approach. There are two 2D tasks, Block Fit and Block Push, and two real robot tasks, screwing hex-screws and opening doors with the 7 DoF Franka Emika Panda arm. We compare both learning performance of the proposed approach against baselines, as well as their ability to generalize to novel environment configurations. To study generalization, we train policies on a small set of training environment configurations and test them on a novel test set. Training over multiple environments is important to avoid overfitting. Details of each task, including controller specifications, task variations, observation and action spaces, and the reward functions can be found in the Appendix.

Block Fit: In this task, a 2D block robot needs to navigate to a 2D goal pose in the scene. There are multiple walls or obstacles in the scene, so the robot cannot directly proceed towards the goal. Figure 5 (Left) shows *some* of the different train and test configurations. The low-level controllers are wall-centric. Different environment configurations have different wall lengths and angles between walls. The training set has 8 different environment configurations, while the test set has 9.

Block Push: In this task, a 2D block robot needs to push another block along a vertical wall over a ledge to a desired goal pose. Figure 5 (right) visualizes *some* train and test configurations. Controllers and environment wall configurations are similar to those of Block Fit. The environment samples the initial pose of the block robot and the target block. The training set has 11 different environment configurations, and the test set has 8.

Franka Hex-Screw: In this task, a 7-DoF Franka Panda arm is used to insert a hex-key into a screw, and turn the screw to a desired angle while applying a downward force and maintaining vertical orientation. The screw will not turn unless a sufficient pre-defined $(20N)$ downward force is applied. Different environment configurations have different wrench and screw sizes. The training set uses size scale multipliers of $(0.9, 1.0, 1.3)$, and the test set uses $(0.7, 0.8, 1.1, 1.2, 1.4, 1.5)$.

Franka Door-Opening: In this task, the Franka robot needs to open a door by first turning its door handle and then pulling the door beyond an opening threshold. To avoid trivial policy solutions, the door will not open unless the handle is first turned to a desired angle. The environment samples the initial relative pose between the EE and the door, and different configurations have different locations of the door handle on the door. The training and test set contain 4 and 3 configurations.

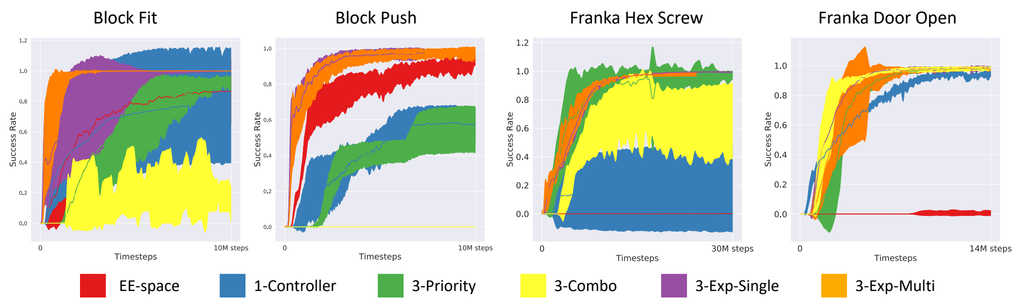


Figure 6: Success rates for all tasks on training environment configurations.

Compared Approaches: We set $N_c = 3$ across all experiments, which we found to be sufficient. To evaluate the utility of our proposed object-axis controllers we compare against an RL agent that controls the robot directly via end-effector delta-poses. We call this approach **EE-Space**. We also evaluate the need for executing multiple controllers in parallel by comparing against a baseline which only chooses 1 controller at each timestep. We call this **1-Ctrlr**. To show the efficacy of our proposed Expanded-MDP formulation we compare against both: discrete combinatorial (**3-Combo**) and continuous priority scores (**3-Priority**) action spaces. Both these approaches naively combine all possible controller combinations and we show how this can lead to sub-optimal performance.

RL Training: We use Proximal Policy Optimization (PPO) [37] implemented in stable-baselines [38] across all tasks and action space variants. Given the high variance in policy-gradient RL algorithms, we run all methods with 8 different seeds (sampled uniformly between 1 and 100). All tasks are simulated with an NVIDIA Isaac Gym [1], a GPU-accelerated robotics simulator [39].

Metrics: We report the success rates of the learned policies separately for train and test environment configurations. Performance on the train set indicates whether or not the approach can robustly solve a task, and performance on the test set evaluates generalization abilities. Test set is split into two subsets, one with small deviations from the train configurations, and another with larger deviations. We report additional results including more fine-grained analysis for each task in the Appendix.

5 Experiment Results and Discussion

Block Tasks: Figure 6 (left) plots the success ratios averaged over all train environment configurations for Block Fit and Block Push. The Expanded-MDP methods are able to successfully learn both tasks. While EE-Space also makes progress on both tasks, it has a lower success rate, and this is due to its inability to robustly solve a few challenging configurations (see Appendix). Both 1-Ctrlr and 3-Priority perform well on Block Fit but poorly on Block Push. We attribute this difference to how there is a greater need to use multiple controllers in the right order for Block Push. For instance, the policy needs to choose a force/position controller that pushes into the wall and then another controller to move up. In addition, robustly pushing the block around the edge of the vertical wall also requires multiple controllers. Although it is feasible to achieve this by quickly switching between controllers, such a strategy is not robust. 1-Ctrlr is unable to use multiple controllers at the same time, and using the high-dimensional priority score action space is challenging.

Table 1 shows success rates for both tasks on two sets of test configurations. Both EE-space and Expanded-MDP methods perform well when test configurations have small deviations from train configurations, with EE-Space performing slightly worse. However, for large deviations, EE-space performs poorly, achieving success ratios of 0.371 for Block Fit and 0.518 for Block Push. By contrast, Expanded-MDP methods perform much better, achieving 0.974 for Block Fit and 0.788 for Block Push, and 3-Priority also outperforms EE-Space for the Block Fit task. In addition, 1-Ctrlr sees greater performance degradation going from small to large deviations in test configurations. Together, these results indicate that using a structured action space of multiple object-centric controllers leads to better generalization than using one controller or directly learning in the EE-space.

Franka Tasks: Figure 6 (right) shows training results for both Franka Hex-Screw and Door-Open tasks. The Expanded-MDP methods perform well on both the tasks, while EE-Space does not make

[1]https://developer.nvidia.com/isaac-gym

Task	Variation	EE-Space	1-Ctrlr	3-Priority	3-Combo	3-Exp-Single	3-Exp-Multi
Block Fit	Train	0.87 (0.213)	0.778 (0.38)	0.936 (0.032)	0.294 (0.18)	0.998 (0.002)	**1.00 (0.0)**
	Test-Small	0.87 (0.10)	0.916 (0.14)	**0.99 (0.001)**	0.184 (0.12)	**0.99 (0.001)**	**0.99 (0.01)**
	Test-Large	0.371 (0.246)	0.396 (0.423)	0.877 (0.141)	0.165 (0.23)	**0.974 (0.048)**	0.953 (0.087)
Block Push	Train	0.966 (0.046)	0.594 (0.087)	0.548 (0.129)	0.0 (0.0)	0.974 (0.025)	**0.978 (0.022)**
	Test-Small	0.912 (0.045)	0.577 (0.193)	0.396 (0.041)	0.0 (0.0)	0.945 (0.045)	**0.960 (0.030)**
	Test-Large	0.518 (0.185)	0.152 (0.137)	0.376 (0.032)	0.0 (0.0)	0.751 (0.103)	**0.788 (0.132)**

Table 1: Mean (SD) success rates for Block Fit and Block Push tasks on different environment configurations.

Task	Variation	EE-Space	1-Ctrlr	3-Priority	3-Combo	3-Exp-Single	3-Exp-Multi
Hex-Screw	Train	0.002 (0.002)	0.183 (0.303)	0.960 (0.048)	0.774 0.194)	**0.984 (0.01)**	0.980 (0.016)
	Test-Small	0.00 (0.00)	0.13 (0.072)	0.62 (0.045)	0.429 (0.430)	0.963 (0.01)	**0.966 (0.015)**
	Test-Large	0.00 (0.00)	0.026 (0.025)	0.633 (0.081)	0.34 (0.057)	**0.936 (0.028)**	**0.936 (0.035)**
	Real-World	n/a	0.0	0.5	0.0	**0.9**	0.6
Door-Open	Train	0.002 (0.006)	0.947 (0.021)	0.982 (0.007)	0.984 (0.013)	**0.987 (0.009)**	0.984 (0.015)
	Test-Small	0.066 (0.063)	0.922 (0.043)	0.965 (0.046)	0.975 (0.011)	**0.997 (0.006)**	0.992 (0.015)
	Test-Large	0.000 (0.001)	0.936 (0.032)	0.983 (0.006)	0.985 (0.007)	**0.996 (0.005)**	0.994 (0.013)
	Real-World	n/a	0.0	**1.0**	0.9	**1.0**	**1.0**

Table 2: Success rates for Franka Hex-Screw and Open-Door tasks on train and test environment configurations across 8 seeds. Parentheses denote standard deviation. Real-world results are evaluated over 10 trials each. We did not run EE-Space policies in the real world as they were unable to learn the tasks in simulation.

progress on either task. For Hex-Screw, the EE-Space policy is able to reach the screw, but is unable to learn to simultaneously rotate the screw and apply sufficient downward force. For Door-Open, the EE-Space policy reaches the door handle, but fails to grasp and completely rotate the door handle in a robust manner to open the door. One reason for these EE-Space failures is that exploration in both tasks is difficult in the end-effector space. To aid EE-Space exploration, we evaluated the approach from [40], which gives the agent additional exploration rewards. While doing so leads the agent to cover a larger region in the state space, the explored states do not always correspond with meaningful behaviors for task completion, so we did not observe any gains using this method.

Unlike with the Block 2D tasks, 3-Priority is able to learn both the Franka tasks. This is because the Franka tasks have fewer possible controllers, which resulted in lower dimensional priority-score action spaces. The reduced action-space dimensions of Franka tasks allowed us to evaluate 3-Combo, which is also able to learn both tasks, although it achieves worse performance on Hex-Screw. Similarly, 1-Ctrlr is able make progress on Door-Open but not Hex-Screw, which suggests that Hex-Screw requires more precise coordination of multiple controllers than Door-Open. Table 2 (rows 2, 3, 5 and 6) shows the success rates for both tasks on test configurations with small and large deviations. All methods that use hierarchical combination of multiple object-axis controllers generalize well to both small and large test deviations. Methods that performed poorly during training, EE-Space for both tasks and 1-Ctrlr Hex-Screw, do not generalize well.

To evaluate Franka tasks in the real-world, we performed 10 trials of each method on the real robot, each trial with a different sampled initial state. For the Hex-Screw task, we further tested on 3 different screw and key sizes. All methods that used the proposed composition of hierarchical controllers were able to robustly perform Door-Open in the real world, while only 3-Exp-Single was able to do so for Hex-Screw. Hex-Screw is more challenging than Door-Open, because it requires more precise movements for alignment and insertion. As a result, sim-to-real gap in the robot dynamics and controller responses leads to greater performance degradation for Hex-Screw than for Door-Open.

6 Conclusion and Future Work

In this work, we propose using RL to learn how to compose hierarchical object-centric controllers for manipulation tasks. Our approach has several advantages. First, the object-centric controllers can be reused across multiple tasks. Second, controller compositions are invariant to certain object properties. Finally, the use of a structured action space introduces meaningful inductive biases for manipulation. Our experiments show that the proposed approach leads to more guided exploration and consequently improved sample efficiency, and it enables zero-shot generalization to test environments and simulation-to-reality transfer without fine-tuning. In future work, we will tackle the main limitations of the current approach – the set of controllers is fixed and manually-defined.

Acknowledgments

This work was supported by NSF Award No. CMMI-1925130, NSF Graduate Research Fellowship Program Grant No. DGE 1745016, Office of Naval Research Grant No. N00014-18-1-2775, ARL grant W911NF-18-2-0218 as part of the A2I2 program, and Nvidia NVAIL.

References

[1] F. J. Abu-Dakka, B. Nemec, J. A. Jørgensen, T. R. Savarimuthu, N. Krüger, and A. Ude. Adaptation of manipulation skills in physical contact with the environment to reference force profiles. *Autonomous Robots*, 39(2):199–217, 2015.

[2] D. H. Ballard. Task frames in robot manipulation. In *AAAI*, volume 19, page 109, 1984.

[3] D. H. Ballard and L. Hartman. Task frames: Primitives for sensory-motor coordination. *Computer Vision, Graphics, and Image Processing*, 36(2-3):274–297, 1986.

[4] M. T. Mason. Compliance and force control for computer controlled manipulators. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(6):418–432, 1981.

[5] M. H. Raibert and J. J. Craig. Hybrid position/force control of manipulators. 1981.

[6] L. Sciavicco and B. Siciliano. *Modelling and control of robot manipulators*. Springer Science & Business Media, 2012.

[7] M. Mühlig, M. Gienger, J. J. Steil, and C. Goerick. Automatic selection of task spaces for imitation learning. In *International Conference on Intelligent Robots and Systems*, pages 4996–5002. IEEE, 2009.

[8] D. Berenson, S. Srinivasa, and J. Kuffner. Task space regions: A framework for pose-constrained manipulation planning. *The International Journal of Robotics Research*, 30(12): 1435–1460, 2011.

[9] J. E. King, M. Cognetti, and S. S. Srinivasa. Rearrangement planning using object-centric and robot-centric action spaces. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3940–3947. IEEE, 2016.

[10] J. Kober, M. Gienger, and J. J. Steil. Learning movement primitives for force interaction tasks. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3192–3199. IEEE, 2015.

[11] A. L. P. Ureche, K. Umezawa, Y. Nakamura, and A. Billard. Task parameterization using continuous constraints extracted from human demonstrations. *IEEE Transactions on Robotics*, 31(6):1458–1471, 2015.

[12] T. Migimatsu and J. Bohg. Object-centric task and motion planning in dynamic environments. *IEEE Robotics and Automation Letters*, 5(2):844–851, 2020.

[13] S. Manschitz, M. Gienger, J. Kober, and J. Peters. Learning sequential force interaction skills. *Robotics*, 9(2):45, 2020.

[14] L. Peternel, L. Rozo, D. Caldwell, and A. Ajoudani. A method for derivation of robot task-frame control authority from repeated sensory observations. *IEEE Robotics and Automation Letters*, 2(2):719–726, 2017.

[15] A. Conkey and T. Hermans. Learning task constraints from demonstration for hybrid force/position control. In *2019 IEEE-RAS 19th International Conference on Humanoid Robots (Humanoids)*, pages 162–169. IEEE, 2019.

[16] O. Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. *IEEE Journal on Robotics and Automation*, 3(1):43–53, 1987.

[17] Y. Nakamura, H. Hanafusa, and T. Yoshikawa. Task-priority based redundancy control of robot manipulators. *The International Journal of Robotics Research*, 6(2):3–15, 1987.

[18] A. Dietrich, C. Ott, and A. Albu-Schäffer. An overview of null space projections for redundant, torque-controlled robots. *The International Journal of Robotics Research*, 2015.

[19] A. Karami, H. Sadeghian, M. Keshmiri, and G. Oriolo. Hierarchical tracking task control in redundant manipulators with compliance control in the null-space. *Mechatronics*, 2018.

[20] R. Martín-Martín, M. Lee, R. Gardner, S. Savarese, J. Bohg, and A. Garg. Variable impedance control in end-effector space. an action space for reinforcement learning in contact rich tasks. In *Proceedings of the International Conference of Intelligent Robots and Systems (IROS)*, 2019.

[21] M. Bogdanovic, M. Khadiv, and L. Righetti. Learning variable impedance control for contact sensitive tasks. *arXiv preprint arXiv:1907.07500*, 2019.

[22] C. C. Beltran-Hernandez, D. Petit, I. G. Ramirez-Alpizar, T. Nishi, S. Kikuchi, T. Matsubara, and K. Harada. Learning contact-rich manipulation tasks with rigid position-controlled robots: Learning to force control. *arXiv preprint arXiv:2003.00628*, 2020.

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

[24] G. Comanici and D. Precup. Optimal policy switching algorithms for reinforcement learning. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 709–714, 2010.

[25] G. D. Konidaris and A. G. Barto. Efficient skill learning using abstraction selection. Citeseer, 2009.

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

[27] P.-L. Bacon, J. Harb, and D. Precup. The option-critic architecture. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[28] Ö. Şimşek, A. P. Wolfe, and A. G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd international conference on Machine learning*, pages 816–823, 2005.

[29] K. Shiarlis, M. Wulfmeier, S. Salter, S. Whiteson, and I. Posner. Taco: Learning task decomposition via temporal alignment for control. *arXiv preprint arXiv:1803.01840*, 2018.

[30] S. Krishnan, R. Fox, I. Stoica, and K. Goldberg. Ddco: Discovery of deep continuous options for robot learning from demonstrations. *arXiv preprint arXiv:1710.05421*, 2017.

[31] M. Sharma, A. Sharma, N. Rhinehart, and K. M. Kitani. Directed-info GAIL: Learning hierarchical policies from unsegmented demonstrations using directed information. In *International Conference on Learning Representations*, 2019.

[32] C. Daniel, H. Van Hoof, J. Peters, and G. Neumann. Probabilistic inference for determining options in reinforcement learning. *Machine Learning*, 104(2-3):337–357, 2016.

[33] R. Liaw, S. Krishnan, A. Garg, D. Crankshaw, J. E. Gonzalez, and K. Goldberg. Composing meta-policies for autonomous driving using hierarchical deep reinforcement learning, 2017.

[34] J. D. Morrow and P. K. Khosla. Manipulation task primitives for composing robot skills. In *Proceedings of International Conference on Robotics and Automation*, volume 4, pages 3354–3359 vol.4, 1997. doi:10.1109/ROBOT.1997.606800.

[35] J. Pazis and M. G. Lagoudakis. Reinforcement learning in multidimensional continuous action spaces. In *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 97–104. IEEE, 2011.

[36] L. Metz, J. Ibarz, N. Jaitly, and J. Davidson. Discrete sequential prediction of continuous actions for deep rl. *arXiv preprint arXiv:1705.05035*, 2017.

[37] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[38] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[39] J. Liang, V. Makoviychuk, A. Handa, N. Chentanez, M. Macklin, and D. Fox. Gpu-accelerated robotic simulation for distributed reinforcement learning. *Conference on Robot Learning (CoRL)*, 2018.

[40] D. Pathak, D. Gandhi, and A. Gupta. Self-supervised exploration via disagreement. In *ICML*, 2019.

[41] K. Zhang, M. Sharma, J. Liang, and O. Kroemer. A modular robotic arm control stack for research: Franka-interface and frankapy. *arXiv preprint arXiv:2011.02398*, 2020.

Appendices

A Controller Implementation Details

A.1 Specific Controllers for each Task

Block Fit. A set of controllers is associated with each wall in the environment. For a wall, let v be the unit vector pointing in the wall's normal direction, x_{wm} be the coordinate of the middle of the wall. The set of controllers associated for each wall include:

1. Position attractor along normal direction. $x_d = x_{wm}$, $u = v$
2. Position attractor along error direction. $x_d = x_{wm}$, $u = \frac{x_d - x_c}{\|x_d - x_c\|_2}$
3. Force attractor along the normal direction. $f_d = 10$, $u = v$
4. Rotation attractor aligning the block's x-axis to the normal. $r_d = v$, $u = [1, 0]^\top$
5. Rotation attractor aligning the block's y-axis to the normal. $r_d = v$, $u = [0, 1]^\top$

Block Push. In addition to all the per-wall controllers of the Block Fit task, Block Push has the following per-wall controllers:

1. Position controller along the side of a wall. Let v' be a unit vector orthogonal to v. Since there are 2 such possible directions, we pick the one that gives the direction pointing up along the vertical wall in the scene.

   Let x_{wc} be the coordinate of a wall corner. Since walls form a corner-connect chain in this task, using one of the two corners per wall covers all corners in the scene except the last corner in the chain, which we ignore.

   With these, this controller has $x_d = x_{wc}$ and $u = v'$.
2. Position curl controller around a wall corner. This controller rotates the end-effector in a fixed-radius circle around a point until it reaches the target position which also lies on the circle. The attractor target is $x_d = x_{wc} + \|x_c - x_{wc}\|_2 v'$, and the direction is $u = R(\frac{\pi}{2})\frac{x_c - x_{wc}}{\|x_c - x_{wc}\|_2}$, where $R(\theta)$ gives a 2D rotation matrix with the angle θ.

Block Push has one more position controller that attracts the robot block toward the target block. Let x_g be the current location of the center of the target block. This position controller has $x_d = x_g$ and $u = \frac{x_d - x_c}{\|x_d - x_c\|_2}$.

Franka Hex-Screw. Let x_s be the location of screw, and $x_g = x_s + [0, 0.02, 0]^\top$ be a point 2cm above the screw (the y-axis is vertical in our coordinate frame). Position attractor controllers use x_g as the target, instead of x_s, because attracting the hex-key tip toward the inside of the screw directly can result in collisions with the side of the screw and prevent the key from properly inserted.

1. Position attractor along vertical direction. $x_d = x_g$, $u = [0, 1, 0]^\top$
2. Position attractor along error direction. $x_d = x_g$, $u = \frac{x_d - x_c}{\|x_d - x_c\|_2}$
3. Position controller that prevents motion in the vertical direction $x_d = x_c$, $u = [0, 1, 0]^\top$. This controller does not attract the end-effector toward a goal. Instead, its utility is solely in its nullspace projection, which ensures lower-priority controllers cannot move the end-effector outside of a horizontal plane. This controller is useful for preventing prematurely inserting the hex-key.
4. Force controller that pushes downward toward the hex screw. $f_d = 20$, $u = [0, -1, 0]^\top$.
5. Rotation controller that maintains the verticality of the end-effector. $r_d = [0, 1, 0]^\top$, $u = [0, 0, 1]^\top$. The positive z-axis of the end-effector frame corresponds to the direction that the hex-key points towards.
6. Rotation controller that rotates the hex-key counter-clockwise. $r_d = R_y(100°)[1, 0, 0]^\top$, $u = [1, 0, 0]^\top$, where $R_y(\theta)$ gives a rotation matrix that rotates around the y-axis with the angle θ.

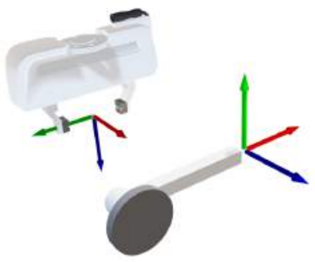


Figure 7: Axes Visualization for Franka End-Effector and Door Handle for the Door-Open Task. RGB corresponds to XYZ.

7. Rotation controller that rotates the hex-key clockwise. $r_d = R_y(-100°)[1,0,0]^\top$, $u = [1,0,0]^\top$

Franka Door-Open. See Figure 7 for a visualization of both the Franka end-effector and door handle axes. Let $r_{\{x,y,z\}}$ correspond to the 3 axes of the door handle. Let x_g be a grasp point on the door handle, x_h be the center of the handle axle (dark gray cylinder in Figure 7). The set of controllers include:

1. Position attractor to door handle along error direction. $x_d = x_g$, $u = \frac{x_d - x_c}{\|x_d - x_c\|_2}$
2. Position curl attractor for rotating around the handle in the plane of the door panel (the nullspace of r_z). Let $x_e = \mathcal{N}(r_z)(x_c - x_h)$. Then $x_d = x_h - \|x_e\|_2 r_y$, $u = \frac{x_e}{\|x_e\|_2} \times r_z$
3. Force controller to pull the handle. $f_d = 50$, $u = -r_z$
4. Rotation controller to align the x-axes of the gripper and the handle. $r_d = r_x$, $u = [1,0,0]^\top$
5. Rotation controller to align the y-axes of the gripper and the handle. $r_d = r_y$, $u = [0,1,0]^\top$
6. Rotation controller to align the z-axes of the gripper and the handle. $r_d = r_z$, $u = [0,0,1]^\top$

A.2 Integral Term for Force Controllers

Using an integral term for force controllers can help reduce the force error and improve stability. Let $\bar{\delta}_f^i$ be the accumulated force errors for the force controller used at the ith priority. Then, the corresponding delta position target is computed as:

$$\Delta_x^i = \mathcal{N}_i(K_f \delta_f(f_d^i, u^i, f_c) + K_I \bar{\delta}_f^i) \tag{8}$$

where $\mathcal{N}_i = \mathcal{N}([u_0, \ldots, u_{i-1}])$.

A.3 Delta Target Magnitude Clipping

To ensure safety and limit the maximum speed at which our controllers can drive the robot, we clip the magnitude of delta position and rotation targets.

Let D_x be the maximum delta translation magnitude corresponding to a position controller, and D_f for a force controller. The clipping for force and position controllers are computed as follows:

$$\Delta_x^i \leftarrow \frac{\min(\|\Delta_x^i\|_2, D_*)}{\|\Delta_x^i\|_2}\Delta_x^i \tag{9}$$

Note that D_* can be D_x or D_f, depending on if the ith controller is a position or force controller.

Similarly, let D_R be the maximum delta rotation angle for rotation controllers:

$$\Delta_R^i \leftarrow \frac{\min(\|\Delta_R^i\|_2, D_R)}{\|\Delta_R^i\|_2}\Delta_R^i \tag{10}$$

A.4 Controller Hyperparameters

Table 3 lists the different hyperparameters used for the object axes-controllers for each task. We list the gains used for each controller as well as the clipping used while executing each controller. Table 4 lists the task-space impedance parameters used for simulation and real-world experiments. We use [41] to implement each controller for real-world experiments.

	Block Fit	Block Push	Franka Hex-Screw	Franka Door-Open
D_x (m)	1	0.5	0.03	0.03
D_f (N)	0.5	0.1	10^{-4}	10^{-4}
D_R (deg)	90	120	10	10
K_x	1	1	1	1
K_f	1	1	1	1
K_I	0	0	10^{-4}	10^{-4}

Table 3: Controller Gains and Magnitude Clips Across Tasks.

	Simulation	Real World
K_S	1000	600
K_D	$2\sqrt{1000}$	$2\sqrt{600}$
T	10	30

Table 4: Task-Space Impedance Control Parameters. K_S is stiffness, K_D is damping, and T is how many timesteps a controller combination runs before the RL policy is queried again. The simulation and real-world values are not the same due to differences in control frequencies and Franka dynamics between real-world and simulation. We tune the real-world values to ensure that the resultant controller behaviors are similar to those in simulation. This tuning was done prior to task evaluations.

B Task Details

B.1 Block Fit

Observations.

1. 2D pose of block robot
2. 2D contact force direction and magnitude experienced by the block robot in the world frame.
3. 2D coordinates of centers and wall corners

Reward Function: Let ϕ_d be the previous distance between the block translation and the goal translation, ϕ_θ be previous the absolute angle difference between the block rotation and the goal rotation, and let ϕ'_d, ϕ'_θ be there current counterparts. The reward function rewards making progress towards the goal with a small alive penalty and a large task completion bonus:

$$R = 10(\phi'_d - \phi_d) + 5(\phi'_\theta - \phi_\theta) - 0.1 + 1000 \times \mathbb{1}(\phi'_d < 0.16 \wedge \phi'_\theta < 5^\circ) \tag{11}$$

The goal translation threshold 0.16 is about half the width of the block.

B.2 Block Push

Observations.

1. 2D pose of block robot
2. 2D contact force direction and magnitude experienced by the block robot in the world frame.

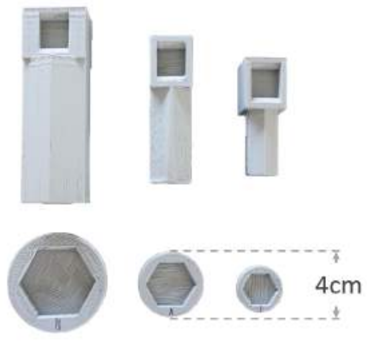


Figure 8: Different hex screw and key sizes used for testing in the real world. The middle size represents $1.0\times$ scale factor, while the left is $1.5\times$, and the right $0.7\times$.

3. 2D coordinates of centers and wall corners
4. 2D pose of the target block

Reward Function: The reward function is similar to that of Block Fit, except the progress rewards are computed w.r.t. the target block, not the block robot, and there is an additional reward term for approaching the target block:

$$R = 10(\phi'_d - \phi_d) + 10(\phi'_b - \phi_b) - 0.1 + 200 \times \mathbb{1}(\phi'_d < 0.05) \tag{12}$$

where ϕ_b is the previous distance between the robot block and the target block, and ϕ'_b is the current counterpart. The goal translation threshold of 0.05 is about half the width of the target block.

B.3 Franka Hex Screw

See Figure 8 for the different screw and key sizes used in real-world experiments.

Observations.

1. 7-dimension robot arm joint angles
2. Gripper width
3. 6D pose of the tip of the hex-screw. Rotations are represented via quaternions
4. End-effector contact forces
5. Position of the hex screw relative to the robot base

Reward Function: Let ϕ_d be the previous distance from the hex-key tip to the hex screw, ϕ_θ be the previous absolute angle difference between the screw angle and its target angle, and let ϕ'_d, ϕ'_θ be their respective current counterparts. Let ρ be the absolute angle difference between the negative y-axis (pointing downwards) and the z-axis of the end-effector. The reward function rewards approaching the hex-screw, making progress in turning the screw, maintaining a vertical hex key orientation, plus a small alive penalty and a large task bonus:

$$R = 400(\phi'_d - \phi_d) + 10(\phi'_\theta - \phi_\theta) - \rho - 1 + 1000 \times \mathbb{1}(\phi'_\theta < 5^\circ) \tag{13}$$

The target screw rotation angle (at which point $\phi_\theta = 0$) is 70°.

B.4 Franka Door Opening

Observations.

1. 7-dimension robot arm joint angles
2. Gripper width
3. 6D pose of the tip of the hex-screw. Rotations are represented via quaternions
4. End-effector contact forces

5. Door panel angle (How much the door has opened, not the angle of the door handle)
6. Position of the door handle relative to the robot base

Reward Function: Let ϕ_d be the previous distance from the end-effector to the door handle grasp point, ϕ_θ be the previous absolute angle difference between the door handle angle and the target handle turning angle, ϕ_ρ be the previous absolute angle difference between the door panel angle and the target door opening angle, and let ϕ'_d, ϕ'_θ, ϕ'_ρ be their respective current counterparts. Let c denote the current end-effector contact forces. The reward function rewards approaching the door handle, turning the handle, turning the door, plus small alive penalties and excessive force penalties, plus a large task bonus:

$$R = 10(\phi'_d - \phi_d) + 10(\phi'_\theta - \phi_\theta) + 100(\phi'_\rho - \phi_\rho) - 0.01 - 0.001\|c\|_2 + 100 \times \mathbb{1}(\phi'_\rho < 5^\circ) \quad (14)$$

The target door handle turning angle (at which point $\phi_\theta = 0$) is 90°, and the target door panel opening angle (at which point $\phi_\rho = 0$) is 35°.

C RL Training Details

C.1 PPO Hyperparameters

	Block Fit	Block Push	Franka Hex-Screw	Franka Door-Open
num steps	500	240	450	480
discount factor	0.995	0.995	0.995	0.995
entropy coefficient	0.01	0.01	$[0.01, 0.1]$	$[0.01, 0.1]$
learning rate	2.5×10^{-4}	2.5×10^{-4}	2.5×10^{-4}	2.5×10^{-4}
value loss coefficient	0.5	0.1	0.5	0.5
max gradient norm	0.5	0.5	0.5	0.5
lambda	0.95	0.95	0.95	0.95
num minibatches	50	30	30	50
num opt epochs	4	4	4	4
clip range	0.2	0.2	0.2	0.2

Table 5: PPO Hyperparameters Across All Tasks.

Table 5 lists the hyperparameters used for each of the experiments. In addition to the above parameters, we also decay the clip range using a linear schedule with a decay rate of 0.99 after every epoch. We set the minimum clip range value to be 0.04. Also, for the Franka Hex-Screw and Franka Door-Open task we evaluated a range of entropy coefficient values $[0.01, 0.1]$ for the end-effector action space.

C.2 Network Architecture

For all tasks and methods, we use the same network architectures for both the policy and value function networks. The network consists of 2 hidden layers with 256 hidden units each.

C.3 Controller Features in the Observation Space

We experimented with giving features of each individual controllers to the RL policy in the Expanded-MDP approach. These features may allow the policy to better reason about the effects of individual controllers, and it also allows the policy to operate on a variable number of controllers. Controller features include a 1-hot encoding of the controller type (position, force or rotation), the controller target, axis, and the current error. We refer to this method as **3-Exp-Features**.

See Figure 9 for an illustration of a policy architecture using controller features. Each controller feature vector c_i of the ith controller (there are N in total) is processed by a shared controller feature encoder. These controller embeddings are then each concatenated with embeddings of the original observations, which include environment observations o and encodings of previously selected controllers c'_j (there are $N_c - 1$ in total). For c'_j, instead of single-1-hot or multi-1-hot embeddings, we

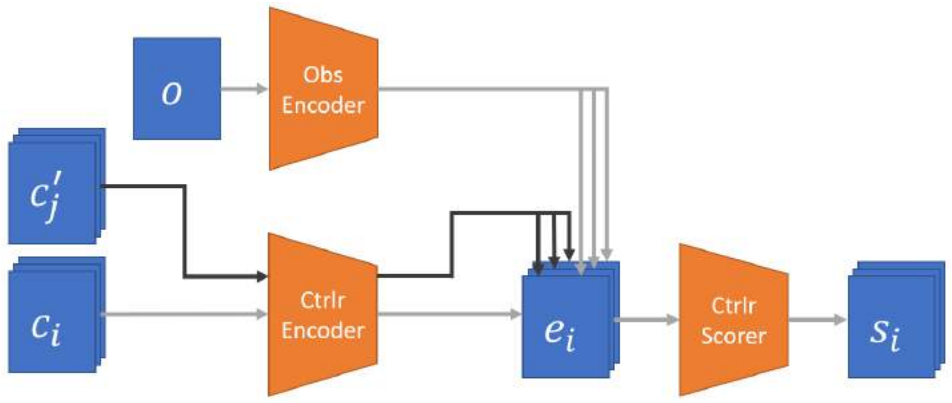


Figure 9: Policy Architecture for 3-Exp-Features.

also use the controller features, which are first processed by the shared controller encoder. Finally, the concatenated embeddings e_i are separately processed and scored by the policy. The normalized scores s_i's form the discrete distribution from which we sample the next controller selection.

Both the observation and controller feature encoders contain two hidden layers of 64 hidden units each. The controller scorer has one hidden layer of 32 hidden units. For the value function, we remove the last linear layer (size 32×1) of the controller scorer in the policy, add the 32-dimensional outputs from the hidden layer across all N intermediate outputs, and pass the sum through one last linear layer (also of size 32×1) to obtain one scalar value.

While 3-Exp-Features policies are invariant to the number of controllers in the scene, we did not explicitly test for this capability. However, we still ran this method alongside the other approaches, and it achieves comparable performance to the other 3-Exp variants (see detailed results below).

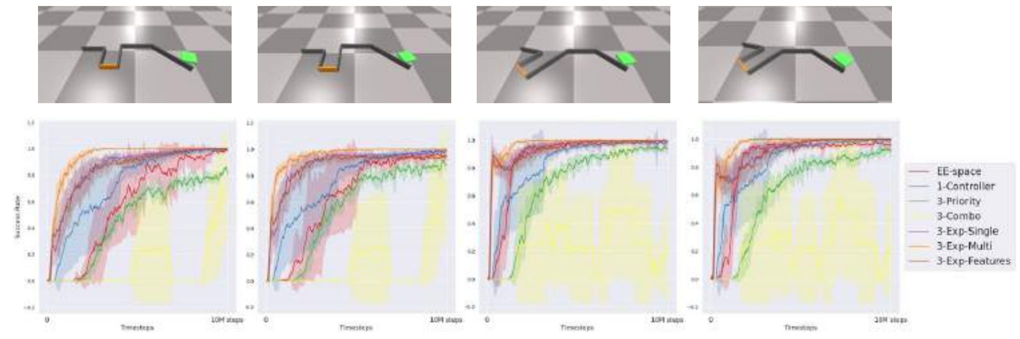
D Detailed Experiment Results

We discuss results for each task in more detail in the following sections. Video results for all the different tasks and methods can be seen at `https://sites.google.com/view/compositional-object-control/`.

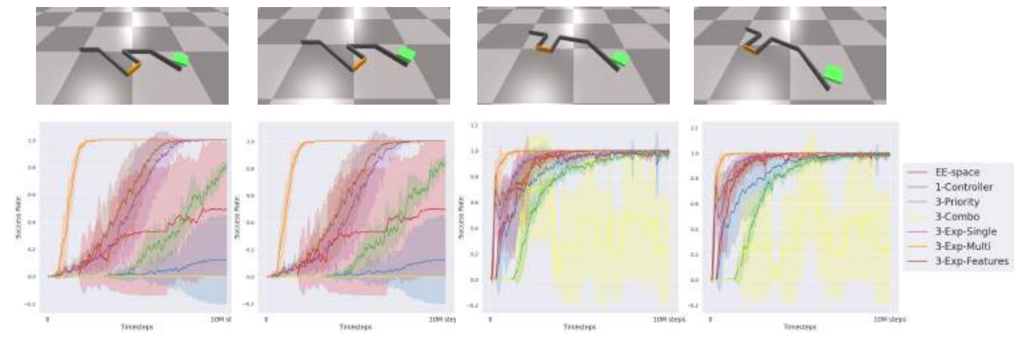
D.1 Block Fit

Figure 10 plots the train results (success-rate) for all different train configurations. As can be seen in the above figure, for all configurations besides a couple (Figure 10b left two plots), all the methods perform quite well during training. The poor performance in two of the training environments is due to their more challenging configurations. In both of these configurations the slot to the target wall is oriented in a different direction, so a robust policy needs to reason about this change. We observe that our proposed Expanded-MDP methods are able to perform well for this configuration. However, the end-effector action space shows a large variance *i.e.* many of the seeds fail to learn a robust policy to solve these configurations. Additionally, we also observe that 1-Ctrlr is unable to solve this task robustly. This shows the advantage of using multiple-controllers in parallel.

Figure 11a shows the different test configurations we evaluated. Each of the test configurations is a delta change in the wall lengths or angles from the train configurations. Table 6 shows the results on each of these test configurations. As seen above, our proposed Expanded-MDP formulations are able to outperform all other methods for all the configurations. 3-Priority performs well on most test configurations besides the slightly harder ones (E, G). This indicates that Expanded-MDP methods are able to learn more robust policies as compared to using a continuous priority score. Additionally, EE-Space perform poorly, especially for more different test configurations (E, F, G, H). Qualitatively, we observe that EE-Space policies often completely fail to generalize to the test configurations. 1-Ctrlr performs poorly on the D, E, and G, H configurations. For the initial two configurations, we observe that the learned policy can often get stuck around wall corners, which prevents it from completing the episode within the given time. Alternately, for the latter two environ-



(a)



(b)

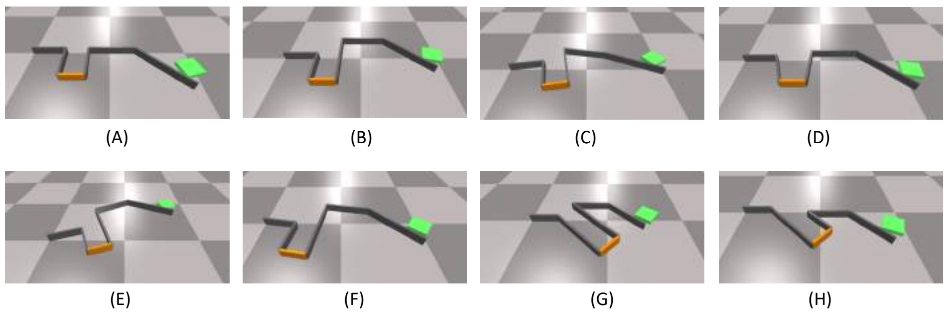
Figure 10: Different environment configurations used to train the Block Fit task. The plot below each environment configuration shows how the trained policy performed on each particular configuration.

ments, the learned policies across all seeds perform quite poorly, so they are not able to generalize to either of the test configurations.

D.2 Block Push

Figure 12 shows the success rate for all the different environment configurations used in the Block Push task. As seen in the above figure, both 1-Ctrlr and 3-Priority show large variance in training performance. This is because both methods fail to learn the task for some seeds. All the Expanded-MDP methods are able to successfully complete the task without large variance. One reason for this is that a robust task strategy requires the use of multiple object-axis controllers to move the object along the vertical wall as well as to move it around the corner of the top wall. Although it is feasible to accomplish the task by quickly switching between controllers, it is hard to find a robust policy relying under such a switching mechanism, especially when controllers are being run for fixed number of steps. Additionally, while EE-Space also solves all the different environment configurations, its sample complexity is worse than the proposed Expanded-MDP methods.

Table 7 evaluates the learned policy on 8 different test configurations. Figure 13a plots each of these test configurations. These test configurations involve small perturbations in either the wall length or the wall angles (or both) from the train configurations. Specifically, we limit small perturbations to be a max change in vertical wall angle of $3 - 5^\circ$ (A, C, D, F), while larger perturbations are sampled from $\sim 6 - 10^\circ$ (B, E, G, H). We observe that EE-Space is usually robust to small perturbations of the environment, while slightly larger perturbations can significantly affect its performance. However, even with small perturbations our expand-MDP based methods are able to outperform the end-effector space. This shows the advantage of using a structured action space for learning, as Expanded-MDP methods perform well across both sets of configurations. Notably, the proposed approach only performs poorly on B and E configurations. For both configurations, as the policy pushes the red block up the middle wall, the agent block (green block) can sometimes end up under the red block, which leads to the green block falling, ending the episode. Additionally, both 1-Ctrlr and 3-Priority perform poorly on the test configurations. This is due to the poor performance of



(a)

Figure 11: Test configurations for the Block Fit task. Table 6 shows results on each environment configuration.

Test-Cfg	EE-Space	1-Ctrl	3-Pri.	3-Combo	3-Exp-Feat.	3-Exp-Single	3-Exp-Multi
A	0.86 (0.07)	0.98 (0.01)	**1.0 (0.0)**	0.19 (0.12)	**1.0 (0.0)**	**1.0 (0.0)**	**1.0 (0.0)**
B	1.0 (0.0)	1.0 (0.0)	**1.0 (0.0)**	0.20 (0.14)	**1.0 (0.0)**	**1.0 (0.0)**	**1.0 (0.0)**
C	0.85 (0.27)	0.96 (0.03)	0.97 (0.03)	0.18 (0.08)	0.98 (0.01)	**0.99 (0.01)**	0.96 (0.01)
D	0.89 (0.14)	0.71 (0.31)	**1.0 (0.03)**	0.165 (0.06)	**1.0 (0.0)**	**1.0 (0.0)**	**1.0 (0.01)**
E	0.16 (0.23)	0.64 (0.16)	0.76 (0.16)	0.0375 (0.05)	0.97 (0.02)	**0.99 (0.01)**	0.99 (0.02)
F	0.75 (0.39)	0.87 (0.16)	**1.0 (0.0)**	0.20 (0.14)	0.98 (0.01)	**1.0 (0.0)**	**1.0 (0.0)**
G	0.17 (0.30)	0.08 (0.23)	0.75 (0.45)	0.02 (0.03)	0.89 (0.13)	0.82 (0.23)	**0.90 (0.15)**
H	0.50 (0.53)	0.0 (0.0)	**1.0 (0.0)**	0.27 (0.18)	**1.0 (0.0)**	**1.0 (0.0)**	**1.0 (0.0)**

Table 6: Block Fit mean success on test environment configurations. Parentheses denote standard deviation across 8 seeds.

some of the learned policies (across a few seeds) on the train configurations. However, good train performance does not necessarily lead to good test performance. Specifically, 1-Ctrlr can move the green block upwards (by using the position controller for the top-wall), but it is often not able to robustly push it around the corner. This shows the advantage of being able to choose multiple object-axis controllers at each step.

D.3 Franka Hex-Screw

Figure 14a plots the mean success rates for all the different approaches (including using controller features) during training. Since performance on all three train configurations (wrench and screw sizes) is very similar, we report one plot which averages the result for all the configurations. As seen in Figure 14a, EE-Space is not able to learn the task. While EE-Space policies can bring the wrench close to the screw, it does not achieve proper alignment and insertion, nor does it apply sufficient downard force, all of which are necessary to accomplish the task. Similarly, 1-Ctrlr also performs poorly. This is expected, since the task requires the use of multiple controllers *i.e.* force or position controller into the screw object while also rotating the wrench simultaneously. For approaches that use multiple object-axis controllers together, we find that the expand-MDP approaches perform the best, robustly learning the task each time. All the other approaches suffer from large variance in task performance.

Table 8 visualizes the result for each of the 6 different test configurations. Each test configuration uses a different wrench and screw scale. Our proposed approach is able to generalize to the different test configurations, achieving ≥ 0.9 success rate for all configurations. Although 3-Priority performs well in training, its test performance is slightly poorer. This is because some of the learned policies (seeds) fail to generalize well to any of the test configurations, while the remaining seeds perform as well as our Expanded-MDP approaches. This variance in performance of the learned policies leads

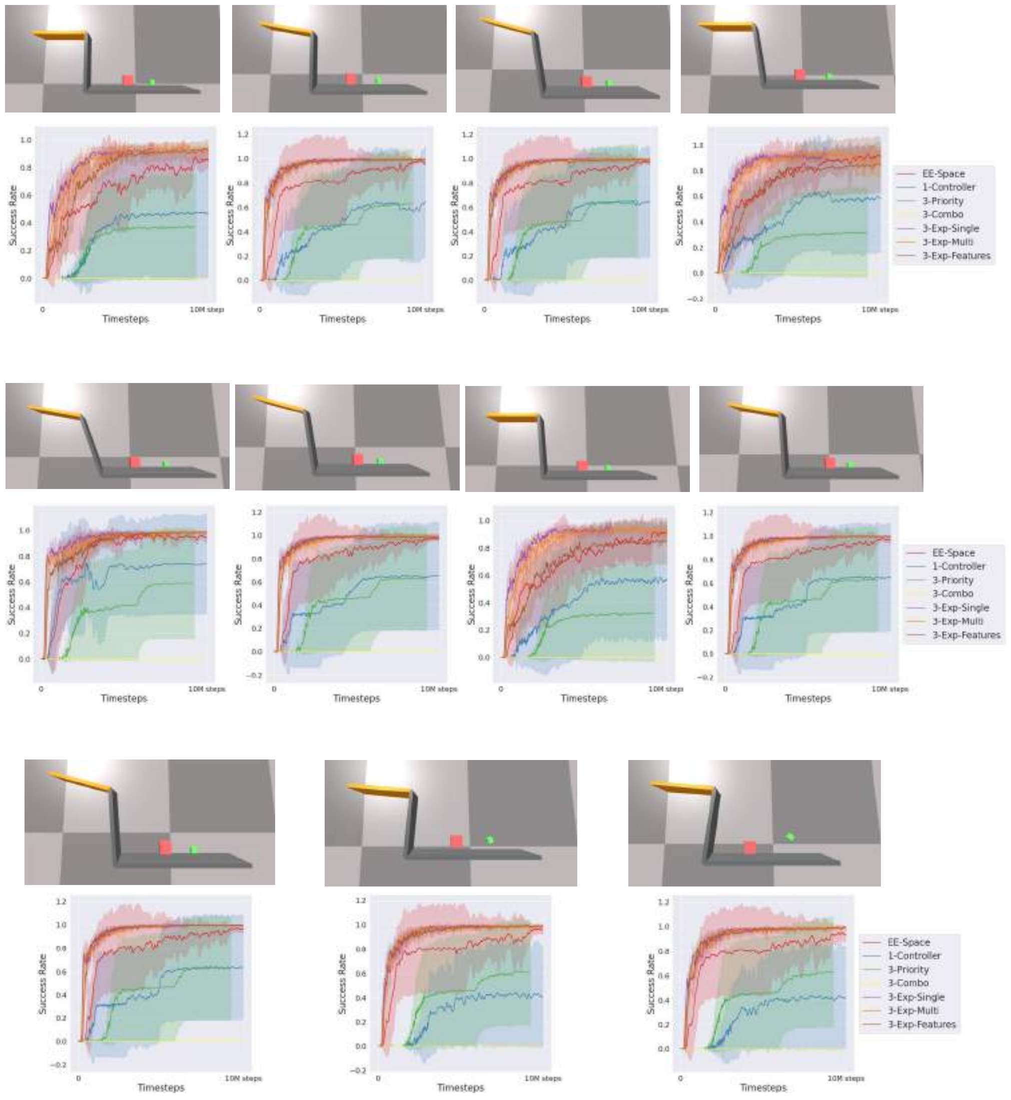
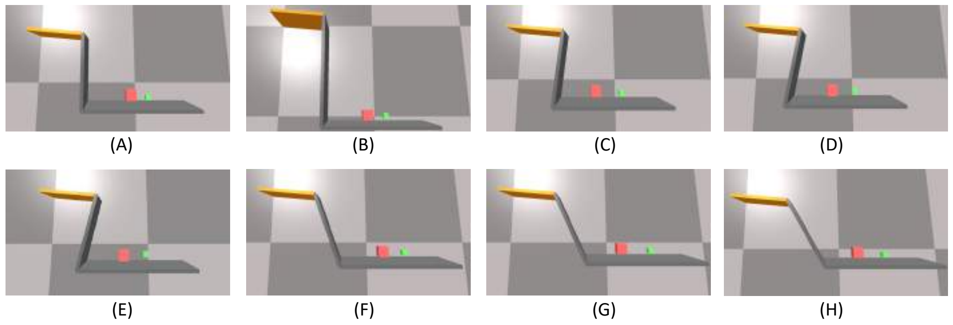


Figure 12: Different environment configurations used to train the Block Push task. The plot below each environment configuration shows how the trained policy performed on each particular configuration.

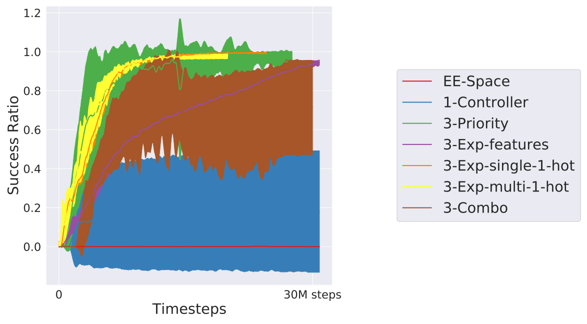


(a)

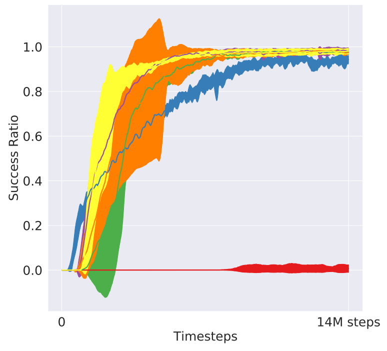
Figure 13: Test environment configurations for the Block Push task. Table 7 shows results on each environment config.

Config	EE-Space	1-Ctrl	3-Priority	3-Combo	3-Exp-Feat.	3-Exp-Single	3-Exp-Multi
A	0.94 (0.06)	0.62(0.47)	0.43 (0.47)	0.0 (0.0)	0.97 (0.02)	0.98 (0.01)	**0.99 (0.00)**
B	0.27 (0.28)	0.27(0.30)	0.38 (0.43)	0.0 (0.0)	0.50 (0.27)	**0.72 (0.18)**	0.65 (0.30)
C	0.86 (0.23)	0.30 (0.40)	0.43(0.37)	0.0 (0.0)	0.91 (0.10)	**0.97 (0.01)**	**0.97 (0.04)**
D	0.70 (0.28)	0.07 (0.13)	0.42 (0.46)	0.0 (0.0)	0.89 (0.06)	**0.93 (0.07)**	0.88 (0.12)
E	0.48 (0.31)	0.01 (0.01)	0.36 (0.39)	0.0 (0.0)	**0.79 (0.17)**	0.69 (0.23)	0.67 (0.17)
F	**0.96 (0.03)**	0.73 (0.41)	0.38 (0.42)	0.0 (0.0)	0.88 (0.11)	0.95 (0.06)	**0.96 (0.03)**
G	0.89 (0.10)	0.67 (0.49)	0.35 (0.39)	0.0 (0.0)	**0.97 (0.03)**	0.92 (0.06)	0.89 (0.07)
H	0.61 (0.26)	0.27 (0.41)	0.34 (0.38)	0.0 (0.0)	0.79 (0.11)	0.78 (0.10)	**0.88** (0.07)

Table 7: Block Push mean success on test environment configurations. Parentheses denote standard deviation across 8 seeds.



(a) Franka Hex-Screw Task



(b) Franka Door-Opening Task

Figure 14: Franka tasks success ratios on training environment configurations during training.

to lower mean success rate for 3-Priority. 1-Ctrlr fails to work well on any of the test configurations, which is expected given its poor training performance.

D.4 Franka Door-Opening

Figure 14b shows the average success rate in all train environments for the Door-Open. All methods except EE-Space are able to learn this task. One reason for this is that object-centric controllers make exploration in this task much more efficient than directly using the end-effector space. Although the EE-Space policy is able to grasp the handle, it fails to turn and pull. Table 9 shows quantitative results on test environments. Methods that use 3 controllers have very similar performance and

Config	EE-Space	1-Ctrl	3-Priority	3-Combo	3-Exp-Feat	3-Exp-Single	3-Exp-Multi
0.7	0.0 (0.00)	0.05(0.14)	0.69 (0.44)	0.45 (0.43)	0.95 (0.04)	**0.97 (0.02)**	0.96 (0.035)
0.8	0.0 (0.00)	0.11(0.17)	0.66 (0.49)	0.43 (0.43)	0.97 (0.05)	0.96 (0.03)	**0.98 (0.01)**
1.1	0.0 (0.00)	0.21(0.37)	0.63 (0.49)	0.43 (0.36)	0.96 (0.03)	**0.98 (0.03)**	0.97 (0.03)
1.2	0.0 (0.00)	0.07 (0.15)	0.57 (0.42)	0.44 (0.42)	**0.97 (0.04)**	0.95(0.03)	0.95 (0.04)
1.4	0.0 (0.00)	0.0 (0.00)	0.67 (0.48)	0.34 (0.36)	0.92 (0.03)	0.94 (0.04)	**0.95 (0.03)**
1.5	0.0 (0.00)	0.03 (0.14)	0.54 (0.46)	0.24 (0.36)	**0.92 (0.04)**	0.90 (0.05)	**0.92 (0.03)**

Table 8: Franka Hex-Screw mean success across all test environment configurations. Parentheses denote standard deviation across 8 seeds.

perform better than 1-Ctrlr. Using multiple controllers is beneficial for this task. When turning the handle, the robot needs to learn to rotate the gripper and press down at the same time; when opening the door, the robot needs to simultaneously press the handle and pull it open. Since the reward function contains separate rewards for approaching the handle, turning the handle, and opening the door, the performance differences are due to the complexity of the task and not a lack of informative reward signals.

Config	EE-Space	1-Ctrl	3-Priority	3-Combo	3-Exp-Feat	3-Exp-Single	3-Exp-Multi
A	0.13 (0.13)	0.87(0.06)	0.93 (0.08)	0.97 (0.01)	**0.99 (0.01)**	**0.99 (0.01)**	**0.99 (0.01)**
B	0.00 (0.00)	0.96 (0.02)	**0.99 (0.01)**	**0.99 (0.01)**	**0.99 (0.01)**	**0.99(0.01)**	**0.99 (0.02)**
C	0.00 (0.00)	0.93 (0.03)	0.99 (0.01)	0.99 (0.01)	**1.00 (0.00)**	0.99 (0.01)	0.99 (0.01)

Table 9: Franka Door-Opening mean success on test environment configurations. Parentheses denote standard deviation across 8 seeds.

E Controller Selection Analyses

We perform an ablation study to better understand the effects of algorithmic choices in our proposed approach. First, we analyze the effects of controller selection frequency, *i.e.*, we analyze the effect of T, where T is the number of steps for which object-axes controllers are run before the RL policy is queried again. Second, we qualitatively evaluate the learned controller selection policy by visualizing the learned policies. For both of these settings we use the Block Fit task.

E.1 Controller Selection Frequency

We evaluate how the controller selection frequency T affects the learning performance. For all previous experiments we use $T = 10$, *i.e.*, the object-axes controllers are run only for a few (10) steps. Although switching controllers frequently allows the RL policy to be more expressive, this comes at the associated cost of higher sample complexity. In this experiment we evaluate learning performance when controllers are allowed to run for much larger steps *i.e.* $T = 80$. To keep the overall simulation time fixed, we simultaneously reduce the maximum number of steps the RL policy is run, *i.e.* we reduce the episode length of the MDP T_{MDP}. This is important since running both the controllers and the RL policy for large number of steps is computationally prohibitive, since the total number of steps taken in the simulator is $T \times T_{\text{MDP}}$. We set $T_{\text{MDP}} = 15$ for this experiment.

Figure 15 plots the average success rate for all train configurations on the Block Fit task with $T = 80$. As seen above, our proposed expand-MDP based approaches are able to perform quite well. Alternately, selecting only one-controller (1-Controller) at each time step performs poorly as compared to $T = 10$ (Figure 10). This shows the advantage of being able to use multiple object-axes controllers in parallel. With small T the 1-Controller policy is able to complete tasks by quickly switching between different controllers. Since this is not possible with a larger T value, its performance decreases. This emphasizes the importance of using multiple-controllers in parallel. Additionally, Figure 15 shows that the Expanded-MDP based approaches are able to learn to perform the task in $30K$ steps only. This is significantly better than the $\sim 10M$ steps required for $T = 10$ (Figure 10).

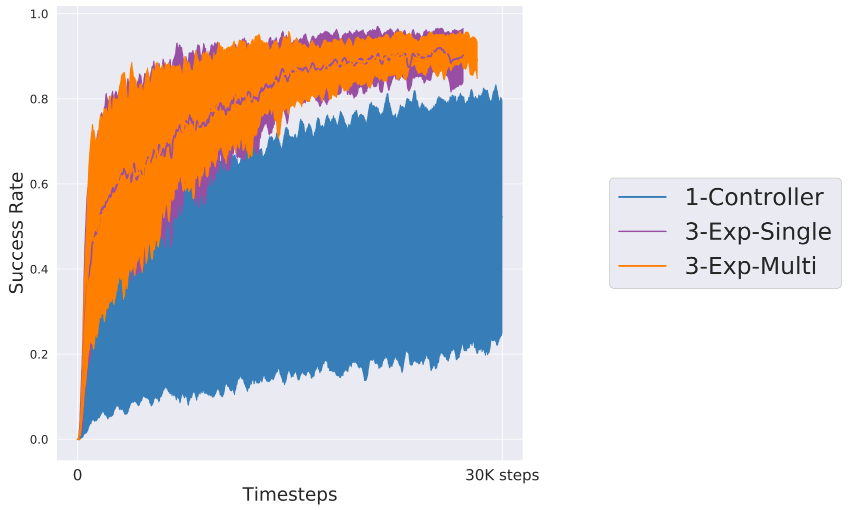


Figure 15: **Controller Selection Frequency:** Success rate for Block Fit task when object axes-controllers are run for $T = 80$ steps. Results averaged over 4 seeds (instead of usual 8).

E.2 Controller Selection Visualization

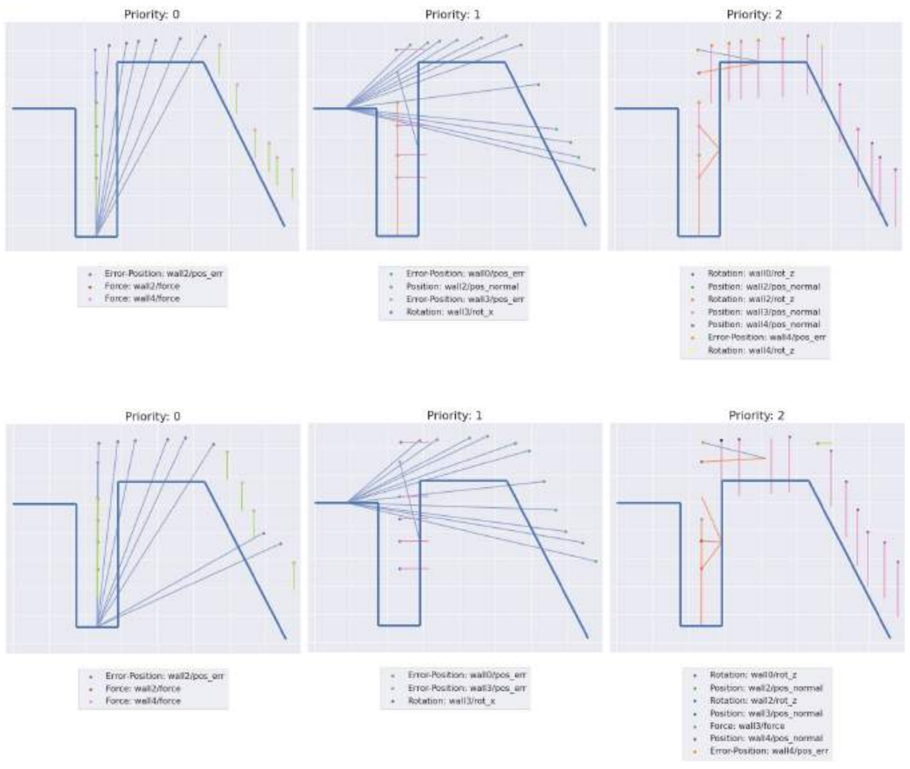

Priority: 0
Priority: 1
Priority: 2
Error-Position: wall2/pos_err
Force: wall2/force
Force: wall4/force
Error-Position: wall0/pos_err
Position: wall2/pos_normal
Error-Position: wall3/pos_err
Rotation: wall3/rot_x
Rotation: wall0/rot_z
Position: wall2/pos_normal
Rotation: wall2/rot_z
Position: wall3/pos_normal
Position: wall4/pos_normal
Error-Position: wall4/pos_err
Rotation: wall4/rot_z
Priority: 0
Priority: 1
Priority: 2
Error-Position: wall2/pos_err
Force: wall2/force
Force: wall4/force
Error-Position: wall0/pos_err
Error-Position: wall3/pos_err
Rotation: wall3/rot_x
Rotation: wall0/rot_z
Position: wall2/pos_normal
Rotation: wall2/rot_z
Position: wall3/pos_normal
Force: wall3/force
Position: wall4/pos_normal
Error-Position: wall4/pos_err

Figure 16: **Controller Selection Visualization** for Block Fit during Task Execution. The thick blue lines show the different walls in the environment. The dots represent the block position at each step. While the arrows represent the wall object used by the selected controller. The left most plot shows the top priority (priority: 0) controller being selected, while the right most plot shows the controllers with lowest priority (priority: 2). Top and bottom rows are two different train configurations (A and B from Figure 11a).

Figure 16 plots the controllers the policy selects along the block trajectory for two different train configuration of Block Fit. For the highest priority controller, the policy tends to select the one that attracts the block toward the target wall. Interestingly, the second priority controllers are associated with a different wall, i.e the left most wall. This shows that the RL policy learns to combine controllers across different objects (walls). For the initial part of the trajectory, the RL policy learns to rotate (priority 0) and move (priority 1) the block simultaneously. This composition of different behaviors is important for the policy to accomplish the task as fast as possible. In addition, the policy chooses from a few set of controllers for both priority 0 and priority 1, while it chooses from a large set of controllers for priority 2. This is because many different choices for the priority 2 controller would often have little to no effect, *e.g.* if both priority 0 and 1 controllers are position or force controllers, then choosing an additional position or force controller for priority 2 will likely have no effect. Thus, it is hard for the policy to learn the appropriate effect for lower priority controllers.