A Appendix

A.1 Pseudocode and an Example

First, we provide pseudocode for the bilevel planning strategy described in the main text.

Algorithm BILEVEL PLANNING WITH NSRTS
- **Input:** NSRT set $\{\langle O, P, E, h, \pi \rangle\}$
- **Input:** Task $\langle s_0, g, H \rangle$
- **Input:** n_{trials}: # of imagined trajectory tries
- `// A* with symbolic components of NSRTs and classical heuristics.`
- $s_0^\uparrow \leftarrow \text{ABSTRACT}(s_0)$
- **for** $\overline{p} \in A^*(s_0^\uparrow, g, H, \{\langle O, P, E, \cdot, \cdot \rangle\})$ **do**
  - **for** n_{trials} *tries* **do**
    - Initialize `plan` as empty list
    - `// Imagine rollout with neural components of ground NSRTs.`
    - $s \leftarrow s_0$
    - **for** *ground NSRT* $\langle \cdot, \cdot, \cdot, \pi, h \rangle \in \overline{p}$ **do**
      - $a \sim \pi(\cdot \mid s)$ `// stochastic`
      - Append a to `plan`
      - $s \leftarrow h(s, a)$
    - **if** $g \subseteq \text{ABSTRACT}(s)$ **then**
      - **return** `plan`

Algorithm 1: Pseudocode for bilevel planning with NSRTs. Inputs are a set of NSRTs and a task (initial state s_0, goal g, and horizon H). The outer loop conducts A* search over the symbolic components of the NSRTs, from the symbolic initial state $s_0^\uparrow = \text{ABSTRACT}(s_0)$ to the symbolic goal g. This A* search produces candidate symbolic plans $\overline{p}$, which are sequences of ground NSRTs. The neural components of these ground NSRTs are used in the inner loop, which tries n_{trials} times to *refine* a symbolic plan into a sequence of continuous actions from the environment action space $\mathcal{A}$. If the goal g holds in the final state, we are done. In practice, we perform an extra optimization (not shown): we terminate the inner loop early whenever $\text{ABSTRACT}(s)$ deviates from the expected states under $\overline{p}$.

Next, we provide pseudocode and an example for the data partitioning algorithm described in the main text. Afterward, we give an example to show how the symbolic learning algorithm would produce NSRT parameters, symbolic preconditions, and symbolic effects.

Algorithm PARTITION TRANSITION DATA
- **Input:** Transition dataset $\mathcal{D} = \{\tau\} = \{(s, a, s')\}$
- Initialize Ψ as empty map
- **for** $\tau \in \mathcal{D}$ **do**
  - **if** *any* `key` *in* Ψ *unifies with* $\text{EFF}(\tau)$ **then**
    - Add τ to partition Ψ[`key`]
  - **else**
    - Initialize partition $\Psi[\text{EFF}(\tau)] = \{\tau\}$
- **return** Ψ

Algorithm 2: Pseudocode for transition data partitioning. Uses subroutine EFF from the main text.

Let us work through an example of the partitioning and symbolic learning algorithms. For the sake of this example, we will work purely with the abstract, predicate-based representations of states, but remember that in practice, these would be abstractions of states that are actually continuous and object-oriented. Consider a dataset containing the following four transitions. where we will leave the continuous actions unspecified ("$\cdot$") for simplicity. Each transition is a tuple containing the abstract state and the abstract next state:

1. $(\{\text{ON}(o_1, o_2), \text{ON}(o_2, o_3)\}, \cdot, \{\text{HOLDING}(o_1), \text{ON}(o_2, o_3)\})$
2. $(\{\text{ON}(o_6, o_7), \text{ON}(o_{12}, o_{13}), \text{ISCLEAN}(o_6)\}, \cdot, \{\text{HOLDING}(o_6), \text{ON}(o_{12}, o_{13}), \text{ISCLEAN}(o_6)\})$
3. $(\{\text{HOLDING}(o_7), \text{ISCLEAN}(o_3), \text{ISWET}(o_7)\}, \cdot, \{\text{ONTABLE}(o_7), \text{ISCLEAN}(o_3), \text{ISWET}(o_7)\})$
4. $(\{\text{HOLDING}(o_4), \text{ISDIRTY}(o_1), \text{ISDRY}(o_4)\}, \cdot, \{\text{ONTABLE}(o_4), \text{ISDIRTY}(o_1), \text{ISDRY}(o_4)\})$

We begin by computing partitions. Recall that $\text{EFF}(\tau)$ is the change in abstract state between s and s'. So, for the first transition, $\text{EFF}(\tau) = \{\text{HOLDING}(o_1), \neg\text{ON}(o_1, o_2)\}$. Since there are no partitions yet, this forms the key to a new partition, containing just the first transition. For the second transition, $\text{EFF}(\tau) = \{\text{HOLDING}(o_6), \neg\text{ON}(o_6, o_7)\}$. Attempting to unify this with the key we added previously is successful: there is a mapping between the two effect sets ($o_1 \leftrightarrow o_6$, $o_2 \leftrightarrow o_7$). So, the second transition is included in the same partition as the first. The $\text{ISCLEAN}(o_6)$ atom does not play a role in this operation since it is not part of the effects (it does not change).

Following a similar process, the third transition is placed into a new partition whose key is $\{\text{ONTABLE}(o_7), \neg\text{HOLDING}(o_7)\}$, and the fourth transition is placed into this same partition due to the mapping $o_7 \leftrightarrow o_4$. The $\text{ISCLEAN}(o_3)$, $\text{ISWET}(o_7)$, $\text{ISDIRTY}(o_1)$, and $\text{ISDRY}(o_4)$ atoms do not play a role in this operation since they are not part of the effects (they do not change).

Now, we are ready to learn the NSRT parameters, preconditions, and effects for this example dataset. Since there are two partitions, we will create two NSRTs, one per partition. Recall that $\text{REF}(\tau)$, in our implementation, is the set of objects appearing in the effects. So, for the first transition we have $\text{REF}(\tau) = \{o_1, o_2\}$, for the second transition we have $\text{REF}(\tau) = \{o_6, o_7\}$, for the third transition we have $\text{REF}(\tau) = \{o_7\}$, and for the fourth transition we have $\text{REF}(\tau) = \{o_4\}$. Since every partition's transitions all have equivalent effects up to object remapping (i.e., we can unify them all), we can simply pick an arbitrary transition from each partition and replace its objects in $\text{REF}(\tau)$ by arbitrary variables to produce the NSRT parameters and effects. For the first partition, the NSRT effects are $\{\text{HOLDING}(?x), \neg\text{ON}(?x, ?y)\}$, and the NSRT parameters are $?x$ and $?y$. For the second partition, the NSRT effects are $\{\text{ONTABLE}(?z), \neg\text{HOLDING}(?z)\}$, and the only NSRT parameter is $?z$. Here, $?x$, $?y$, and $?z$ are variables that can stand in for any possible object; in practice, we also check object types (not included in this example) within the implementation of unification, so that these variables can themselves be typed for added efficiency.

To calculate NSRT preconditions, we must first compute the *projected* abstract state for each transition.

(Note that the abstract *next* state in each transition does not play a role in precondition computation.) The PROJECT operation removes any atoms from a state that contain an object *not* mentioned in $\text{REF}(\tau)$. For the first transition's abstract state, PROJECT produces $\{\text{ON}(o_1, o_2)\}$. For the second transition's abstract state, PROJECT produces $\{\text{ON}(o_6, o_7), \text{ISCLEAN}(o_6)\}$. For the third transition's abstract state, PROJECT produces $\{\text{HOLDING}(o_7), \text{ISWET}(o_7)\}$. For the fourth transition's abstract state, PROJECT produces $\{\text{HOLDING}(o_4), \text{ISDRY}(o_4)\}$.

We produce the NSRT preconditions for the first partition (first and second transitions) by substituting in the parameters and taking an intersection: $\{\text{ON}(x, y)\} \cap \{\text{ON}(x, y), \text{ISCLEAN}(x)\} = \{\text{ON}(x, y)\}$. Similarly for the second partition (third and fourth transitions): $\{\text{HOLDING}(z), \text{ISWET}(z)\} \cap \{\text{HOLDING}(z), \text{ISDRY}(z)\} = \{\text{HOLDING}(z)\}$. Notice that for the first NSRT, because o_6 was clean in the second transition but o_1 was not clean in the first transition, ISCLEAN is not included in the preconditions. Similarly, for the second NSRT, because o_7 was wet in only the third transition, and o_4 was dry in only the fourth transition, neither ISWET nor ISDRY is included in the preconditions.

A.2 Handling Failures

Handling Failures in Planning. Here we describe how a failure prediction model can be used to optimize the planning method outlined in Section 5. In the paragraph below, we describe how to learn this model. The reason that this planning procedure is external to the rest of planning with NSRTs is that it uniquely involves propagating information from continuous planning back to symbolic planning. The procedure is a simplified domain-independent version of the domain-dependent error propagation method used in the popular TAMP system of Srivastava et al. (2014). Following Srivastava et al. (2014), we begin by making a crucial assumption: whenever *fail* is reached, the environment reports a set of objects $\{o_1, \ldots, o_j\}$ that were *involved* in the failure (e.g., two objects that are in collision, or an object that broke irreparably). We introduce special predicates NOTCAUSESFAILURE for every object type in the environment, and for each NSRT, we add a symbolic effect $\text{NOTCAUSESFAILURE}(o_i)$ for each o_i in the parameters O. This says that every action affecting a set of objects absolves all those objects from being responsible for a failure; we found this simple technique to be sufficient for our experimental domains, but other, more domain-specific information can be leveraged instead. Finally, during refinement of a symbolic plan, if a failure is predicted at any timestep (see next paragraph), we immediately terminate the inner loop, update the preconditions of the ground NSRT at that timestep to include $\{\text{NOTCAUSESFAILURE}(o_1), \ldots, \text{NOTCAUSESFAILURE}(o_j)\}$ (where $\{o_1, \ldots, o_j\}$ are the set of objects predicted to be involved in the failure, under the learned model described in the next paragraph), and restart A* from the initial state. Effectively, this change forces the planner to either consider actions which change the states of these objects before using the same ground NSRT, or just avoid using this ground NSRT entirely.

Learning to Predict Failures. Here we address the problem of learning to anticipate failures during planning. Note that unlike NSRT learning (Section 6), which is "locally scoped" to a fixed number of objects defined by the NSRT parameters, failure prediction can require reasoning about all objects in the full state. Recall our assumption that the environment reports a set of objects $\{o_1, \ldots, o_j\}$ that were *involved* in failures; so, using the transitions that resulted in *fail*, we can create a dataset of the form $\{(s, a, \mathcal{O}_{\textit{fail}})\}$, where $\mathcal{O}_{\textit{fail}}$ is the set of objects involved in each failure. On this data, we train a graph neural network that takes as input s, $\text{ABSTRACT}(s)$, and a, and outputs a score between 0 and 1 for each object, representing the predicted probability that it is included in $\mathcal{O}_{\textit{fail}}$. Graph neural networks are well-suited to this type of reasoning, because they are relational and can reason about continuous-valued dependencies. We create one node in the graph for each object in the task; the feature vector of each node includes the object's attribute values and arity-1 ground atoms in $\text{ABSTRACT}(s)$. Edges between nodes correspond to arity-2 ground atoms in $\text{ABSTRACT}(s)$; higher-arity predicates can be converted into arity-2 ones. For the output graph, each node has a single feature corresponding to the score. Once trained, we use this model to predict the failure set by including all objects whose score is over 0.5.

A.3 Extended Environment Details

Environment 1: In "PickPlace1D," a robot must pick blocks and place them into designated target regions on a table. All pick and place poses lie along a 1D axis. There are three **object types**: blocks, targets, and obstructors. (The robot is abstracted away for simplicity.) Blocks have one **attribute**: a 1D pose. Targets have two: a start pose and an end pose. Obstructors have three: a start pose and an end pose, along with a third attribute indicating orthogonal distance from the 1D axis. **Actions** are 2D, with the first dimension representing a pose at which to execute a pick, and the second dimension representing a pose at which to place. Each action updates the state of at most one block or obstructor according to whether the pick pose is within a small tolerance of the object's pose. Placing a block within some tolerance of an obstructor results in a collision. Picking and placing an obstructor always moves the obstructor away from the 1D axis, preventing future collisions. The **behavior prior** randomly chooses to pick obstructors or pick blocks that are not yet at their target region, and then place them away (for obstructors) or on a random target region (for blocks). There are three **predicates**: ON(?BLOCK, ?TARGET), INFREESPACE(?BLOCK), and ISREMOVED(?OBSTRUCTION), with the semantics suggested by the names. Across all tasks, blocks start in free space, obstructors are each initially obstructing some target region, and goals are to move each block to be ON a unique target. **Training tasks** feature 2 or 5 blocks, 5 or 10 targets, and 0 or 1 obstructors, and have horizon $H = 10$. **Easy test tasks** feature 2 blocks, 5 targets, and 0 or 1 obstructors, and have horizon $H = 25$. **Hard test tasks** feature 4 blocks, 12

	PickPlace1D		Kitchen		Blocks		Painting	
Methods	Easy	Hard	Easy	Hard	Easy	Hard	Easy	Hard
Bilevel planning with NSRTs (Ours)	2.826	15.492	0.599	2.736	1.414	3.935	0.696	8.731
Bilevel planning with prior (B6)	3.140	8.425	9.778	10.295	5.510	10.735	15.882	0.331
Forward shooting with prior (B7)	4.106	0.000	0.000	1.323	4.807	2.736	3.276	0.000

Table 2: This table, referenced in Appendix A.5, is a companion to Table 1 in the main text, showing the numerical standard deviations of the means. See Table 1 caption in Section 7 for details.

targets, and 2 obstructors, and have horizon $H = 25$.

Environment 2: In "Kitchen," a robot waiter must pick cups, fill them with water, wine, or coffee, and serve them to customers. There are three **object types**: cups, customers, and robots. Cup **attributes** include 6D pose, mass, what liquid is in the cup (an integer indicating empty, water, wine, or coffee), whether the cup has been served (true or false), and whether or not the cup is currently held by a robot (true or false). Customer attributes include an integer ID and current drink. The singular robot attribute is a 1D gripper joint state. **Actions** are 5D: the first three dimensions represent the xyz pose of a cup to be picked, the fourth dimension represents the ID of a customer to be served, and the fifth represents a liquid to be poured. There are no robot trajectories in this environment; we simply assume kinematic feasibility for every action. Given an action, if the xyz pose is close enough to an unserved cup, and that cup is not too heavy, the cup is picked; otherwise, if the customer ID matches that of some customer and a cup is currently held, the held cup is delivered to the corresponding customer; otherwise, if the liquid is close enough to water, wine, or coffee, and if a cup is held, then the respective liquid is poured into the cup. If the robot tries to pick up a cup that is too heavy, no change occurs in the environment. The **behavior prior** randomly picks cups, pours liquids, or serves cups to unserved customers. The **predicates** are: CUSTOMERHASCOFFEE(?CUSTOMER), CUSTOMERHASWATER(?CUSTOMER), CUSTOMERHASWINE(?CUSTOMER), GRIPPEROPEN(?ROBOT), HOLDING(?CUP), CUPUNSERVED(?CUP), CUPHASCOFFEE(?CUP), CUPHASWATER(?CUP), CUPHASWINE(?CUP). Across all tasks, there is only one robot; customers are initially unserved and cups are initially empty; and goals involve the CUSTOMERHASCOFFEE, CUSTOMERHASWATER, and CUSTOMERHASWINE predicates. **Training tasks** feature 2 or 3 cups and 1 customer, and have horizon $H = 10$. **Easy test tasks** feature 2 cups and 1 customer, and have horizon $H = 3$. **Hard test tasks** feature 3 cups and 2 customers, and have horizon $H = 6$.

Environment 3: In "Blocks," modeled after the classic AI blocksworld domain, a robot must stack blocks on a table to make towers. There are two **object types**: blocks and robots. Block **attributes** include a 3D pose, whether or not the block is held (true or false), and whether or not the block has another block above it (true or false). Robot attributes include a 1D gripper joint state. **Actions** are 4D, with the dimensions representing target end effector xyz pose and target gripper joint state. A position controller is used to navigate the end effector to the target pose. When an action is taken, if the target end effector pose is close enough to a block, that block is clear from above, the target gripper state is open enough, and no other block is held, then the block is picked. If a block is already held, and the target end effector pose is close enough to a clear location on the table, then the block is placed on the table at that location; if, instead, the target end effector pose is close enough to a clear block, then the held block is stacked on top of the clear block. The **behavior prior** randomly picks a block, or attempts to place a block on the table or another block. The **predicates** are: ON(?BLOCK1, ?BLOCK2), ONTABLE(?BLOCK), GRIPPEROPEN(?ROBOT), HOLDING(?BLOCK), CLEAR(?BLOCK). Across all tasks, there is only one robot, and goals involve the ON predicate. **Training tasks** feature 3 or 4 blocks, and have horizon $H = 20$. **Easy test tasks** feature 3 blocks, and have horizon $H = 25$. **Hard test tasks** feature 5 or 6 blocks, and have horizon $H = 35$.

Environment 4: In "Painting," a robot must pick, wash, dry, paint, and place widgets into a box or shelf. Placing into the box requires picking with a top grasp; placing into the shelf requires picking with a side grasp. The box has a lid that may obstruct placements; whether the lid will obstruct a placement is not represented symbolically. This environment was introduced by Silver et al. (2021). There are five **object types**: widgets, boxes, lids, shelves, and robots. Widget **attributes** include 3D pose, 1D color, 1D wetness, 1D dirtiness, and whether or not the widget is held (true or false). Box and shelf attributes include only a 1D color. Lid attributes are 1D, indicating the degree to which the lid is open. Robot attributes include a 1D end effector rotation (modulating between top and side grasps) and a 1D gripper joint state. **Actions** are 8D: the first four dimensions are target end effector pose and rotation, the fifth dimension is target gripper joint state, the sixth dimension is a "water level," the seventh dimension is a "heat level," and the final dimension is a color for painting. A position controller is used to navigate the end effector to the target pose and rotation. Actions with high enough water or heat levels wash or dry a held widget, respectively; actions with paint colors that are close enough to either the shelf or box color result in painting the held object that color; otherwise, the action results in a pick, a place, or no effect, depending on whether an object is currently held, the target gripper state is near enough to either "open" or "closed," and whether the current end effector rotation matches the requirements of the desired placement (box placements require top grasps; shelf placements require side grasps). Picking a box lid has the effect of opening it. The **behavior**

prior randomly picks, washes, dries, paints, or places objects. The **predicates** are: OnTable, Holding, HoldingSide, HoldingTop, InShelf, InBox, IsDirty, IsClean, IsDry, IsWet, IsBlank, IsShelfColor, IsBoxColor, all parameterized by a single ?widget, and GripperOpen(?robot). Across all tasks, there is only one robot, box, lid, and shelf, and the goal is to paint each widget a certain color (each box or shelf color) and place it in the corresponding receptacle. **Training tasks** feature 2 or 3 widgets, and have horizon $H = 18$. **Easy test tasks** feature 1 widget, and have horizon $H = 6$. **Hard test tasks** feature 10 widgets, and have horizon $H = 60$.

A.4 Extended Method Details

Here, we provide additional details about the methods.

The NSRT action samplers and low-level transition models are always fully connected neural networks with hidden layer sizes [32, 32]. All neural networks are trained using the Adam optimizer (Kingma and Ba 2014) for 35K (action samplers), 10K (low-level transition models), or 50K (applicability classifier) epochs with a learning rate of 1e-3.

In our robotic environments of interest, transitions are often *sparse*, changing only a subset of object attributes at any given time. For learning the low-level transition model, we exploit this by calculating the attributes that change in *any* transition within a partition, and only predict next values for those attributes, leaving the others unchanged.

For learning the action samplers, we restrict the covariance matrix Σ to be diagonal and positive semi-definite using an exponential linear unit (Clevert, Unterthiner, and Hochreiter 2016). During evaluation only, we clip samples from the action samplers to be at most 1 standard deviation from the mean, for improved stability.

The applicability classifier is also a fully connected neural network with hidden layer sizes [32, 32]. We train it with negative examples collected from either other partitions, or data from the same partition but with the objects re-mapped. We subsample negative examples to ensure that the dataset is balanced, in a 1:1 ratio, with the positive examples.

In all experiments, we use $n_{\text{trials}} = 1$, which we found to be sufficient due to the accuracy of the action samplers and low-level transition models.

For implementing the h_{add} heuristic, we use the Pyperplan (Alkhazraji et al. 2020) software package.

All GNNs (both the failure predictor, and the action-value function of B4) are standard encode-process-decode architectures (Battaglia et al. 2018), where node and edge modules are fully connected neural networks with one hidden layer of dimension 16, ReLU activations, and layer normalization. Message passing is performed for $K = 3$ iterations. Training uses the Adam optimizer (Kingma and Ba 2014) for 500 epochs with learning rate 1e-3 and batch size 128. For the action-value function, we train by running 5 iterations of fitted Q-iteration, and during evaluation, we sample 100 candidate actions from the behavior prior π_0 at each step, choosing the action with the best predicted value to execute in the environment.

Methods that use shooting (B2 and B7) try up to 1000 iterations, or until the timeout (3 seconds for every method across all experiments) is reached. Methods that perform rejection sampling from the behavior prior (B6 and B7) with the learned applicability classifiers try up to 30 times before giving up and returning a random action from the behavior prior.

Figure 3 shows that B4 (the action-value function learning baseline) performs very poorly. In preliminary experiments, we had verified that it works in much easier test task instances than were used for any of our main results in Figure 3. The main finding from those preliminary experiments was that action-value function learning requires a lot more data than we are working with in this paper; B4 began to perform at the level of our approach given about 2000 training episodes, on those very easy test task instances (whereas our main results are only conducted up to 500 training episodes). This finding is consistent with the general principle that model-free learning strategies are known to be data-hungry (Moerland, Broekens, and Jonker 2020).

A.5 Ablation Standard Deviation Results

Table 2 reports the standard deviations for the ablation experiments, accompanying the means shown in Table 1. They were omitted from Table 1 due to space reasons. See Section 7 for details.

Thank you all for your time and helpful comments!

R1 asks about "the exact relation to methods in robotics planning." In this work, we do not innovate on methods for robotic TAMP; instead, we commit to a particular class of TAMP (search-then-sample), then illustrate how NSRTs can be learned and used for planning within this class. **R1** also asks about "the influence of the quality of the learned models on planning." We agree that robustness studies would be informative; see also our response to R3. Finally, as **R1** suggests, we will definitely emphasize novelty in the writing. To our knowledge, we are the first to learn operator structure, samplers, and transition models in one system. Learning all three represents a significant advance toward scaling TAMP with minimal reliance on expert input. Moreover, designing a representation where each of these pieces can be learned, and where the fundamental leverage available in TAMP can be exploited during planning, is nontrivial; prior works on learning-for-TAMP often have discordant assumptions and perspectives, which prevents a straightforward integration.

R2: Goals are specified in the symbolic language defined by the provided predicates, e.g. if given a predicate `On`, a goal might be `On(block1, block2)`. This type of goal enables symbolic planning. We did not use an off-the-shelf planner here because we need to continue searching if the first symbolic plan found is unable to be refined; it is not easy to continue in this way with planners like Fast Downward. Regarding Blocks: our approach does not make the downward refinability assumption, but can certainly work when the assumption does hold. We included Blocks to validate our hypothesis that the advantage of our approach over baseline B1 is due to not assuming downward refinability. Regarding the "dependency of the proposed method on the given priors," the priors must produce data sufficient for symbolic learning, which requires receiving enough "counterexamples" (transitions where an action failed to achieve an effect due to failing preconditions, e.g. picking up an object while the gripper is already holding something else).

R3's major concerns revolve around the unrealistic assumptions in the simulator. We are indeed not attempting to model contact dynamics, which would be very challenging for a NN transition model to capture (especially in a data-efficient manner). So, our environments involve tolerances such as the "close enough" condition (with a very small ϵ) for grasping objects, to avoid the zero-measure sampling problem which often afflicts learning-for-TAMP systems. Nevertheless, we disagree with **R3** that "the continuous part is not relevant for most actions": it is extremely important that the sampler network learns to output actions within this tolerance, otherwise planning will often fail. One of the major ideas of this paper is that learning a *separate* sampler network per NSRT, which is specific to *that NSRT's* symbolic preconditions and effects, makes learning data-efficient and decreases the complexity of the true function each NN is attempting to approximate. **R3** also suggests that our work cannot be "adapted to a physically accurate simulation and even less so to its usage in real-life." Although NN transition models typically have difficulty with realistic contacts, this does not make them unadaptable to such settings. In real-life problems, we expect that NSRTs can be used in settings where either (1) zero-measure manipulation is not required (for instance, grasping need not be perfect, and placing need not be precisely aligned with the table top), or (2) the neural samplers' and dynamics' outputs can be used in conjunction with a local optimizer to make fine adjustments. **R3** also asks about B6 on Easy tasks. In Easy tasks, rejection sampling from the behavior prior will sometimes succeed within the sampling budget. For example, if the behavior prior randomly grasps objects, then rejection sampling towards grasping a specific object will be more likely to succeed when there are fewer objects overall. This naive sampling can be made arbitrarily bad by decreasing the sampling budget or introducing more objects, as seen in the Hard tasks. In contrast, the learned NSRTs generate values conditioned on the NSRT effects — for example, grasps for a specific object based on its geometry. **R3** is also concerned about handling failures with a large number of objects. Note that even in problems where several objects must be removed to clear a path, the planner will typically encounter a collision, plan to remove that collision, and repeat; at each stage, there will be only one additional object in the NSRT. This part of planning can be understood as a generalization of Stilman's NAMO algorithm ("Manipulation Planning Among Movable Obstacles", ICRA 2007).

R4 comments that "replacing the motion planner (with a learning-based approach) within prior frameworks for TAMP is often a superficial change." This comment suggests a possible misunderstanding; from a TAMP perspective, we are learning operators, samplers, and transition models. Unlike a motion planner, these components are domain-specific and must be hand-specified for every new TAMP domain, which motivates our learning-based approach. **R4** asks why A* was used for symbolic planning. We also experimented with GBFS, but found in preliminary work that A* performs better. See the R2 response for more details on off-the-shelf symbolic planners, and note that state-of-the-art planners such as Fast Downward similarly use A* or GBFS. **R4** asks about dimensionality; see Appendix A.3. **R4** asks about learning and planning time in the experiments. Here are total learning times (in seconds) for our main method after 100 training episodes, with (mean, stdev) over 5 seeds:

- PickPlace1D: (148.7, 4.4)
- Kitchen: (331.6, 26.1)
- Blocks: (225.8, 12.0)
- Painting: (684.2, 18.8)

Planning times averaged over all test problems:

- PickPlace1D: Easy: (2.0, 0.3); Hard: (22.5, 2.3)
- Kitchen: Easy: (61.7, 0.7); Hard: (95.0, 14.4)
- Blocks: Easy: (1.3, 0.1); Hard: (22.4, 3.3)
- Painting: Easy: (9.6, 5.0); Hard: (181.0, 41.7)

We will include these results in the next version of the paper.

R4 asks about conditions under which an NSRT can be reused. This is a very interesting question, and has connections to transfer in deep model-based RL. In our setting, two conditions must hold for NSRTs to be reusable: (1) the symbolic preconditions and operators must continue to describe the abstract transition model of this new task, and (2) the NNs must make predictions on data from the same distribution they were trained on.