Predicate Invention for Bilevel Planning

Tom Silver[*1], **Rohan Chitnis**[*2], **Nishanth Kumar**[1], **Willie McClinton**[1],
Tomás Lozano-Pérez[1], **Leslie Pack Kaelbling**[1], **Joshua B. Tenenbaum**[1]

[1]MIT Computer Science and Artificial Intelligence Laboratory [2]Meta AI
{tslvr, njk, wbm3, tlp, lpk, jbt}@mit.edu, ronuchit@meta.com

Abstract

Efficient planning in continuous state and action spaces is fundamentally hard, even when the transition model is deterministic and known. One way to alleviate this challenge is to perform bilevel planning with abstractions, where a high-level search for abstract plans is used to guide planning in the original transition space. Previous work has shown that when state abstractions in the form of symbolic predicates are hand-designed, operators and samplers for bilevel planning can be learned from demonstrations. In this work, we propose an algorithm for learning predicates from demonstrations, eliminating the need for manually specified state abstractions. Our key idea is to learn predicates by optimizing a surrogate objective that is tractable but faithful to our real efficient-planning objective. We use this surrogate objective in a hill-climbing search over predicate sets drawn from a grammar. Experimentally, we show across four robotic planning environments that our learned abstractions are able to quickly solve held-out tasks, outperforming six baselines.

1 Introduction

Hierarchical planning is a powerful approach for decision-making in environments with continuous states, continuous actions, and long horizons. A crucial bottleneck in scaling hierarchical planning is the reliance on human engineers to manually program domain-specific abstractions. For example, in bilevel sample-based task and motion planning (Srivastava et al. 2014; Garrett et al. 2021), an engineer must design (1) symbolic predicates; (2) symbolic operators; and (3) samplers that propose different refinements of the symbolic operators into continuous actions. However, recent work has shown that when predicates are *given*, operators and samplers can be learned from a modest number (50–200) of demonstrations (Silver et al. 2021; Chitnis et al. 2022). Our objective in this work is to *learn* predicates that can then be used to learn operators and samplers.

Predicates in bilevel planning represent a discrete state abstraction of the underlying continuous state space (Li, Walsh, and Littman 2006; Abel, Hershkowitz, and Littman 2017). For example, `On(block1, block2)` is an abstraction that discards the exact continuous poses of `block1` and `block2`. State abstraction alone is useful for decision-making, but predicates go further: together with operators, predicates enable the use of highly-optimized domain-independent AI planners (Helmert 2006).

We consider a problem setting where a small set of *goal predicates* are available and sufficient for describing task goals, but practically insufficient for bilevel planning. For example, in a block stacking domain (Figure 1), we start with `On` and `OnTable`, but have no predicates for describing whether a block is currently held or graspable. Our aim is to invent new predicates to enrich the state abstraction beyond what can be expressed with the goal predicates alone, leading to stronger reasoning at the abstract level.

What objective should we optimize to learn predicates for bilevel planning? First, consider our real objective: we want a predicate set such that bilevel planning is fast and successful, in expectation over a task distribution, when we use those predicates to learn operators and samplers for planning. Unfortunately, this real objective is far too expensive to use directly, since even a single evaluation requires neural network sampler training and bilevel planning.

In this work, we propose a novel surrogate objective that is deeply connected to our real bilevel-planning objective, but is tractable for predicate learning. Our main insight is that demonstrations can be used to analytically approximate bilevel planning time. To leverage this objective for predicate learning, we take inspiration from the program synthesis literature (Menon et al. 2013; Ellis et al. 2020), and learn predicates via a hill-climbing search through a grammar, with the search guided by the objective. After predicate learning, we use the predicates to learn operators and samplers. All three components can then be used for efficient bilevel planning on new tasks.

In experiments across four robotic planning environments, we find predicates, operators, and samplers learned from 50–200 demonstrations enable efficient bilevel planning on held-out tasks that involve different numbers of objects, longer horizons, and larger goal expressions than seen in the demonstrations. Furthermore, predicates learned with our proposed surrogate objective substantially outperform those learned with objectives inspired by previous work, which are based on prediction error (Pasula, Zettlemoyer, and Kaelbling 2007; Jetchev, Lang, and Toussaint 2013), bisimulation (Konidaris, Kaelbling, and Lozano-Perez 2018;

[*]These authors contributed equally.

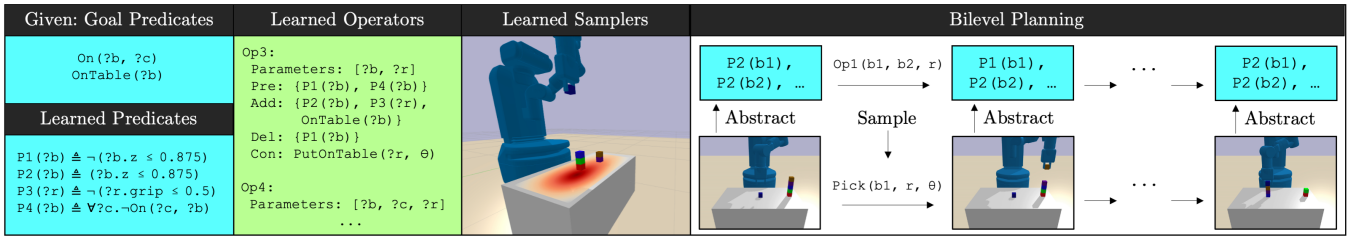


Figure 1: **Overview of our framework.** Given a small set of goal predicates (first panel, top), we use demonstration data to learn new predicates (first panel, bottom). In this Blocks example, the learned predicates `P1` – `P4` intuitively represent `Holding`, `NotHolding`, `HandEmpty`, and `NothingAbove` respectively. Collectively, the predicates define a state abstraction that maps continuous states x in the environment to abstract states s. Object types are omitted for clarity. After predicate invention, we learn abstractions of the continuous action space and transition model via planning operators (second panel). For each operator, we learn a sampler (third panel), a neural network that maps continuous object features in a given state to continuous action parameters for controllers which can be executed in the environment. In this example, the sampler proposes different placements on the table for the held block. With these learned representations, we perform bilevel planning (fourth panel), with search in the abstract spaces guiding planning in the continuous spaces.

Curtis et al. 2021), and inverse planning (Baker, Saxe, and Tenenbaum 2009; Ramírez and Geffner 2010; Zhi-Xuan et al. 2020). We compare against several other baselines and ablations of our system to further validate our results.

2 Problem Setting

We consider learning from demonstrations in deterministic planning problems. These problems are goal-based and object-centric, with continuous states and hybrid discrete-continuous actions. Formally, an *environment* is a tuple $\langle \Lambda, d, \mathcal{C}, f, \Psi_G \rangle$, and is associated with a distribution $\mathcal{T}$ over *tasks*, where each task $T \in \mathcal{T}$ is a tuple $\langle \mathcal{O}, x_0, g \rangle$.

Λ is a finite set of object *types*, and the map $d : \Lambda \to \mathbb{N}$ defines the dimensionality of the real-valued feature vector for each type. Within a task, $\mathcal{O}$ is an *object set*, where each object has a type drawn from Λ; this $\mathcal{O}$ can (and typically will) vary between tasks. $\mathcal{O}$ induces a state space $\mathcal{X}_\mathcal{O}$ (going forward, we simply write $\mathcal{X}$ when clear from context). A *state* $x \in \mathcal{X}$ in a task is a mapping from each $o \in \mathcal{O}$ to a feature vector in $\mathbb{R}^{d(\text{type}(o))}$; x_0 is the initial state of the task.

$\mathcal{C}$ is a finite set of *controllers*. A controller $C((\lambda_1, \ldots, \lambda_v), \Theta) \in \mathcal{C}$ can have both discrete typed parameters $(\lambda_1, \ldots, \lambda_v)$ and a continuous real-valued vector of parameters Θ. For instance, a controller `Pick` for picking up a block might have one discrete parameter of type `block` and a Θ that is a placeholder for a specific grasp pose. The controller set $\mathcal{C}$ and object set $\mathcal{O}$ induce an action space $\mathcal{A}_\mathcal{O}$ (going forward, we write $\mathcal{A}$ when clear). An *action* $a \in \mathcal{A}$ in a task is a controller $C \in \mathcal{C}$ with both discrete and continuous arguments: $a = C((o_1, \ldots o_v), \theta)$, where the objects $(o_1, \ldots o_v)$ are drawn from the object set $\mathcal{O}$ and must have types matching the controller's discrete parameters $(\lambda_1, \ldots, \lambda_v)$. Transitions through states and actions are governed by $f : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$, a known, deterministic transition model that is shared across tasks.

A *predicate* ψ is characterized by an ordered list of types $(\lambda_1, \ldots, \lambda_m)$ and a lifted binary state classifier $c_\psi : \mathcal{X} \times \mathcal{O}^m \to \{\text{true}, \text{false}\}$, where $c_\psi(x, (o_1, \ldots, o_m))$ is defined only when each object o_i has type λ_i. For instance, the predicate `Holding` may, given a state and two objects, robot and block, describe whether the block is held by the robot in this state. A *lifted atom* is a predicate with typed variables (e.g., `Holding(?robot, ?block)`). A *ground atom* $\underline{\psi}$ consists of a predicate ψ and objects $(o_1, \ldots, o_m)$, again with all type$(o_i) = \lambda_i$ (e.g., `Holding(robby, block7)`). Note that a ground atom induces a binary state classifier $c_{\underline{\psi}} : \mathcal{X} \to \{\text{true}, \text{false}\}$, where $c_{\underline{\psi}}(x) \triangleq c_\psi(x, (o_1, \ldots, o_m))$.

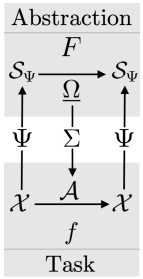
Ψ_G is a small set of *goal predicates* that we assume are given and sufficient for representing task goals, but insufficient practically as standalone state abstractions. Specifically, the goal g of a task is a set of ground atoms over predicates in Ψ_G and objects in $\mathcal{O}$. A goal g is said to *hold* in a state x if for all ground atoms $\underline{\psi} \in g$, the classifier $c_{\underline{\psi}}(x)$ returns true. A solution to a task is a *plan* $\pi = (a_1, \ldots, a_n)$, a sequence of actions $a \in \mathcal{A}$ such that successive application of the transition model $x_i = f(x_{i-1}, a_i)$ on each $a_i \in \pi$, starting from x_0, results in a final state x_n where g holds.

The agent is provided with a set of *training tasks* from $\mathcal{T}$ and a set of demonstrations $\mathcal{D}$, with *one demonstration per task*. We assume action costs are unitary and demonstrations are near-optimal. Each demonstration consists of a training task $\langle \mathcal{O}, x_0, g \rangle$ and a plan π^* that solves the task. Note that for each π^*, we can recover the associated state sequence starting at x_0, since f is known and deterministic. The agent's objective is to *efficiently* solve held-out tasks from $\mathcal{T}$ using anything it chooses to learn from $\mathcal{D}$.

3 Predicates, Operators, and Samplers

Since the agent has access to the transition model f, one approach for optimizing the objective described in Section 2 is to forgo learning entirely, and solve any held-out task by running a planner over the state state $\mathcal{X}$ and action space $\mathcal{A}$. However, searching for a solution directly in these large spaces is highly infeasible. Instead, we propose to *learn abstractions* using the provided demonstrations. In this section, we will describe representations that allow for fast bilevel planning with abstractions (Section 4). In Section 5, we then describe how to learn these abstractions.

We adopt a very general definition of an abstraction (Konidaris and Barto 2009): mappings from $\mathcal{X}$ and $\mathcal{A}$ to alternative state and action spaces. We first characterize an abstract state space $\mathcal{S}_\Psi$ and a transformation from states in $\mathcal{X}$ to abstract states. Next, we describe an abstract action space $\underline{\Omega}$ and an abstract transition model $F : \mathcal{S}_\Psi \times \underline{\Omega} \to \mathcal{S}_\Psi$ that can be used to plan in the abstract space. Finally, we define samplers Σ for refining abstract actions back into $\mathcal{A}$, i.e., actions that can be executed. See the diagram on the right for a summary.



(1) An abstract state space. We use a set of predicates Ψ (as defined in Section 2) to induce an abstract state space $\mathcal{S}_\Psi$. Recalling that a ground atom $\underline{\psi}$ induces a classifier $c_{\underline{\psi}}$ over states $x \in \mathcal{X}$, we have:

Definition 1 (Abstract state)**.** *An* abstract state *s is the set of ground atoms under Ψ that hold true in x:*

$$s = \textsc{Abstract}(x, \Psi) \triangleq \{\underline{\psi} : c_{\underline{\psi}}(x) = \text{true}, \forall \psi \in \Psi\}.$$

The (discrete) abstract state space induced by Ψ is denoted $\mathcal{S}_\Psi$. Throughout this work, we use predicate sets Ψ that are supersets of the given goal predicates Ψ_G. However, only the goal predicates are given, and they alone are typically very limited; in Section 5, we will discuss how the agent can use data to *invent predicates* that will make up the rest of Ψ. See Figure 1 (first panel) for an example.

(2) An abstract action space and abstract transition model. We address both by having the agent learn *operators*:

Definition 2 (Operator)**.** *An* operator *is a tuple $\omega = \langle \textsc{Par}, \textsc{Pre}, \textsc{Eff}^+, \textsc{Eff}^-, \textsc{Con} \rangle$ where:*

- Par *is an ordered list of* parameters*: variables with types drawn from the type set Λ.*
- Pre, Eff$^+$, Eff$^-$ *are* preconditions*,* add effects*, and* delete effects*, each a set of lifted atoms over Ψ and* Par*.*
- Con *is a tuple $\langle C, \textsc{Par}_{\textsc{Con}} \rangle$ where $C((\lambda_1, \ldots, \lambda_v), \Theta)$ is a controller and $\textsc{Par}_{\textsc{Con}}$ is an ordered list of* controller arguments*, each a variable from* Par*. Furthermore, $|\textsc{Par}_{\textsc{Con}}| = v$, and each argument i must be of the respective type λ_i.*

We denote the set of operators as Ω. See Figure 1 (second panel) for an example. Unlike in STRIPS (Fikes and Nilsson 1971), our operators are augmented with controllers and controller arguments, which will allow us to connect to the task actions in **(3)** below. Now, given a task with object set $\mathcal{O}$, the set of all *ground operators* defines our (discrete) abstract action space for a task:

Definition 3 (Ground operator / abstract action)**.** *A* ground operator *$\underline{\omega} = \langle \omega, \delta \rangle$ is an operator ω and a substitution δ : Par $\to \mathcal{O}$ mapping parameters to objects. We use $\underline{\textsc{Pre}}, \underline{\textsc{Eff}^+}, \underline{\textsc{Eff}^-}$, and $\underline{\textsc{Par}_{\textsc{Con}}}$ to denote the ground preconditions, ground add effects, ground delete effects, and ground controller arguments of $\underline{\omega}$, where variables in* Par *are substituted with objects under δ.*

We denote the set of ground operators (the abstract action space) as $\underline{\Omega}$. Together with the abstract state space $\mathcal{S}_\Psi$, the preconditions and effects of the operators induce an abstract transition model for a task:

Definition 4 (Abstract transition model)**.** *The* abstract transition model *induced by predicates Ψ and operators Ω is a partial function $F : \mathcal{S}_\Psi \times \underline{\Omega} \to \mathcal{S}_\Psi$. $F(s, \underline{\omega})$ is only defined if $\underline{\omega}$ is* applicable *in s: $\underline{\textsc{Pre}} \subseteq s$. If defined, $F(s, \underline{\omega}) \triangleq (s - \underline{\textsc{Eff}^-}) \cup \underline{\textsc{Eff}^+}$.*

(3) A mechanism for refining abstract actions into task actions. A ground operator $\underline{\omega}$ induces a partially specified controller, $C((o_1, \ldots o_v), \Theta)$ with $(o_1, \ldots o_v) = \underline{\textsc{Par}_{\textsc{Con}}}$, where object arguments have been selected but continuous parameters Θ have not. To *refine* this abstract action $\underline{\omega}$ into a task-level action $a = C((o_1, \ldots o_v), \theta)$, we use *samplers*:

Definition 5 (Sampler)**.** *Each operator $\omega \in \Omega$ is associated with a* sampler *$\sigma : \mathcal{X} \times \mathcal{O}^{|\textsc{Par}|} \to \Delta(\Theta)$, where $\Delta(\Theta)$ is the space of distributions over Θ, the continuous parameters of the operator's controller.*

Definition 6 (Ground sampler)**.** *For each ground operator $\underline{\omega} \in \underline{\Omega}$, if $\underline{\omega} = \langle \omega, \delta \rangle$ and σ is the sampler associated with ω, then the* ground sampler *associated with $\underline{\omega}$ is a state-conditioned distribution $\underline{\sigma} : \mathcal{X} \to \Delta(\Theta)$, where $\underline{\sigma}(x) \triangleq \sigma(x, \delta(\textsc{Par}))$.*

We denote the set of samplers as Σ. See Figure 1 (third panel) for an example.

What connects the transition model f, abstract transition model F, and samplers Σ? While previous works enforce the downward refinability property (Marthi, Russell, and Wolfe 2007; Pasula, Zettlemoyer, and Kaelbling 2007; Jetchev, Lang, and Toussaint 2013; Konidaris, Kaelbling, and Lozano-Perez 2018), it is important in robotics to be robust to violations of this property, since learned abstractions will typically lose critical geometric information. Therefore, we only require our learned abstractions to satisfy the following *weak semantics*: for every ground operator $\underline{\omega}$ with partially specified controller $C((o_1, \ldots, o_v), \Theta)$ and associated ground sampler $\underline{\sigma}$, there exists some $x \in \mathcal{X}$ and some θ in the support of $\underline{\sigma}(x)$ such that $F(s, \underline{\omega})$ is defined and equals s', where $s = \textsc{Abstract}(x, \Psi)$, $a = C((o_1, \ldots, o_v), \theta)$, and $s' = \textsc{Abstract}(f(x, a), \Psi)$. Note that downward refinability (Marthi, Russell, and Wolfe 2007) makes a much stronger assumption: that this statement holds for *every* $x \in \mathcal{X}$ where $F(s, \underline{\omega})$ is defined.

4 Bilevel Planning

To use the components of an abstraction — predicates Ψ, operators Ω, and samplers Σ — for efficient planning, we build on *bilevel* planning techniques (Srivastava et al. 2014; Garrett et al. 2021). We conduct an outer search over *abstract plans* using the predicates and operators, and an inner search over refinements of an abstract plan into a task solution π using the predicates and samplers.

Definition 7 (Abstract plan)**.** *An* abstract plan *$\hat{\pi}$ for a task $\langle \mathcal{O}, x_0, g \rangle$ is a sequence of ground operators $(\underline{\omega}_1, \ldots, \underline{\omega}_n)$ such that applying the abstract transition*

PLAN(x_0, g, Ψ, Ω, Σ)
// Parameters: $n_{\texttt{abstract}}$, $n_{\texttt{samples}}$.
1 $s_0 \leftarrow$ ABSTRACT(x_0, Ψ)
2 **for** $\hat{\pi}$ *in* GENABSTRACTPLAN(s_0, g, Ω, $n_{abstract}$)
3 **if** $\pi \sim$ REFINE($\hat{\pi}$, x_0, Ψ, Σ, $n_{samples}$) **then**
4 **return** π

Algorithm 1: Pseudocode for our bilevel planning algorithm. The inputs are an initial state x_0, goal g, predicates Ψ, operators Ω, and samplers Σ; the output is a plan π. An outer loop runs GENABSTRACTPLAN, which generates plans in the abstract state and action spaces. An inner loop runs REFINE, which attempts to refine each abstract plan $\hat{\pi}$ into a plan π. If REFINE succeeds, then the found plan π is returned as the solution; if REFINE fails, then GENABSTRACTPLAN continues.

model $s_i = F(s_{i-1}, \underline{\omega}_i)$ *successively starting from* $s_0 =$ ABSTRACT(x_0, Ψ) *results in a sequence of abstract states* $(s_0, \ldots, s_n)$ *that achieves the goal, i.e.,* $g \subseteq s_n$*. This* $(s_0, \ldots, s_n)$ *is called the* expected abstract state sequence.

Because downward refinability does not hold in our setting, an abstract plan $\hat{\pi}$ is *not* guaranteed to be refinable into a solution π for the task, which necessitates bilevel planning. We now describe the planning algorithm in detail.

The overall structure of the planner is outlined in Algorithm 1. For the outer search that finds abstract plans $\hat{\pi}$, denoted GENABSTRACTPLAN (Alg. 1, Line 2), we leverage the STRIPS-style operators and predicates (Fikes and Nilsson 1971) to automatically derive a domain-independent heuristic popularized by the AI planning community, such as LMCut (Helmert and Domshlak 2009). We use this heuristic to run an A* search over the abstract state space $\mathcal{S}_\Psi$ and abstract action space $\underline{\Omega}$. This A* search is used as a generator (hence the name GENABSTRACTPLAN) of abstract plans $\hat{\pi}$, outputting one at a time[1]. Parameter n_{abstract} governs the maximum number of abstract plans that can be generated before the planner terminates with failure.

For each abstract plan $\hat{\pi}$, we conduct an inner search that attempts to REFINE (Alg. 1, Line 3) it into a solution π (a plan that achieves the goal under the transition model f). While various implementations of REFINE are possible (Chitnis et al. 2016), we follow Srivastava et al. (2014) and perform a backtracking search over the abstract actions $\underline{\omega}_i \in \hat{\pi}$. Recall that each $\underline{\omega}_i$ induces a partially specified controller $C_i((o_1, \ldots, o_v)_i, \Theta_i)$ and has an associated ground sampler $\underline{\sigma}_i$. To begin the search, we initialize an indexing variable i to 1. On each step of search, we sample continuous parameters $\theta_i \sim \underline{\sigma}_i(x_{i-1})$, which fully specify an action $a_i = C_i((o_1, \ldots, o_v)_i, \theta_i)$. We then check whether $x_i = f(x_{i-1}, a_i)$ obeys the expected abstract state sequence, i.e., whether $s_i =$ ABSTRACT(x_i, Ψ). If so, we continue on to $i \leftarrow i+1$. Otherwise, we repeat this step, sampling a new

[1]This usage of A* search as a generator is related to top-k planning (Katz et al. 2018; Ren, Chalvatzaki, and Peters 2021). We experimented with off-the-shelf top-k planners, but chose A* because it was faster in our domains. Note that the abstract plan generator is used heavily in learning (Section 5).

ETPT(x_0, g, Ψ, Ω, π^*)
// Note: does not take in samplers!
// Parameters: $n_{\texttt{abstract}}$, t_{upper}.
1 $s_0 \leftarrow$ ABSTRACT(x_0, Ψ)
2 $p_{\text{terminate}} \leftarrow 0.0$
3 $t_{\text{expected}} \leftarrow 0.0$
4 **for** $\hat{\pi}$ *in* GENABSTRACTPLAN(s_0, g, Ω, $n_{abstract}$)
5 $p_{\text{refined}} \leftarrow$ ESTIMATEREFINEPROB$(\hat{\pi}, \pi^*)$
6 $p_{\text{terminate}} \leftarrow (1 - p_{\text{terminate}}) \cdot p_{\text{refined}}$
7 $t_{\text{iter}} \leftarrow$ ESTIMATETIME$(\hat{\pi}, x_0, \Psi, \Omega)$
8 $t_{\text{expected}} \leftarrow t_{\text{expected}} + p_{\text{terminate}} \cdot t_{\text{iter}}$
9 $t_{\text{expected}} \leftarrow t_{\text{expected}} + (1 - p_{\text{terminate}}) \cdot t_{\text{upper}}$
10 **return** t_{expected}

Algorithm 2: Pseudocode for Estimate Total Planning Time in our predicate invention surrogate objective. Commonalities with Algorithm 1 are shown in blue. See Section 5 for details.

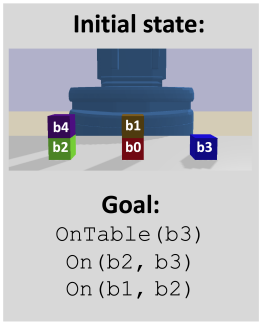
$\theta_i \sim \underline{\sigma}_i(x_{i-1})$. Parameter n_{samples} governs the maximum number of times we invoke the sampler for a single value of i before backtracking to $i \leftarrow i-1$. REFINE succeeds if the goal g holds when $i = |\hat{\pi}|$, and fails when i backtracks to 0.

If REFINE succeeds given a candidate $\hat{\pi}$, the planner terminates with success (Alg. 1, Line 4) and returns the plan $\pi = (a_1, \ldots, a_{|\hat{\pi}|})$. Crucially, if REFINE fails, we continue with GENABSTRACTPLAN to generate the next candidate $\hat{\pi}$. In the taxonomy of task and motion planners (TAMP), this approach is in the "search-then-sample" category (Srivastava et al. 2014; Dantam et al. 2016; Garrett et al. 2021). As we have described it, this planner is *not* probabilistically complete, because abstract plans are not revisited. Extensions to ensure completeness are straightforward (Chitnis et al. 2016), but are not our focus in this work.

5 Learning from Demonstrations

To use bilevel planning at evaluation time, we must learn predicates, operators, and samplers at training time. We use the methods of Chitnis et al. (2022) for operator learning and sampler learning; see Section A.1 and Section A.3 for descriptions. For what follows, it is important to understand that operator learning is fast ($O(|\mathcal{D}|)$), but sampler learning is slow, and both require a given set of predicates. Our main contribution is a method for predicate invention that precedes operator and sampler learning in the training pipeline.

Inspired by prior work (Bonet and Geffner 2019; Loula et al. 2019; Curtis et al. 2021), we approach the predicate invention problem from a program synthesis perspective (Stahl 1993; Lavrac and Dzeroski 1994; Cropper and Muggleton 2016; Ellis et al. 2020). First, we define a compact representation of an infinite space of predicates in the form of a *grammar*. We then enumerate a large pool of *candidate predicates* from this grammar, with simpler candidates enumerated first. Next, we perform a *local search* over subsets of candidates, with the aim of identifying a good final subset to use as Ψ. The crucial question in this step is: what *objective function* should we use to guide the search over candidate predicate sets?



Search Iteration	Predicate Set	J_{surr} (lower is better)	Success Rate on Evaluation Tasks
0	`On(?b, ?c)` `OnTable(?b)`	$1.3 \cdot 10^7$	0%

1	`On(?b, ?c)`
	`OnTable(?b)`
	`¬(?b.z ≤ 0.875)`
		$1.0 \cdot 10^7$	12%

2	`On(?b, ?c)`
	`OnTable(?b)`
	`¬(?b.z ≤ 0.875)`
	`∀?c.¬On(?c, ?b)`
		$2.0 \cdot 10^6$	14%

3	`On(?b, ?c)`
	`OnTable(?b)`
	`¬(?b.z ≤ 0.875)`
	`∀?c.¬On(?c, ?b)`
	`¬(?r.grip ≤ 0.5)`
		9351	100%

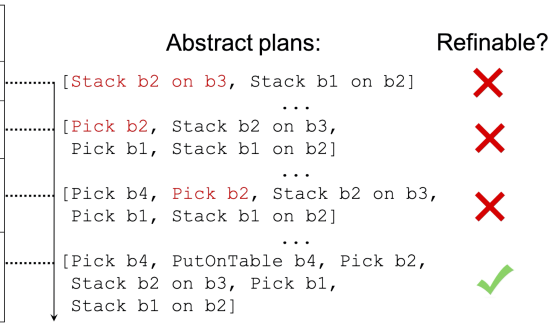


Figure 2: **Predicate invention via hill climbing**. (Left) An example task in Blocks. (Middle) Hill climbing over predicate sets, starting with the goal predicates Ψ_G. On each iteration, the single predicate that improves J_{surr} the most is added to the set. The rightmost table column shows success rates on held-out evaluation tasks. Each iteration of hill climbing adds a predicate that causes all abstract plans above the dotted line to be pruned from consideration. At iteration 0, the robot believes it can achieve the goal by simply stacking `b2` on `b3` and `b1` on `b2`, even though it hasn't picked up either block. The first step of this abstract plan (shown in red) is thus unrefinable. At iteration 1, a predicate with the intuitive meaning `Holding` is added, which makes the A* only consider abstract plans that pick up blocks before stacking them. Still, the abstract plan shown is unrefinable on the first step because `b4` is obstructing `b2` in the initial state. At iteration 2, a predicate with the intuitive meaning `NothingAbove` is added, which allows the agent to realize that it must move `b4` out of the way if it wants to pick up `b2`. This plan is still unrefinable, though: the second step fails, because the abstraction still does not recognize that the robot cannot be holding two blocks simultaneously. Finally, at iteration 3, a predicate with the intuitive meaning `HandEmpty` is added, and planning succeeds.

5.1 Scoring a Candidate Predicate Set

Ultimately, we want to find a set of predicates Ψ that will lead to efficient planning, after we use the predicates to learn operators Ω and samplers Σ. I.e., our real objective is:

$$J_{\text{real}}(\Psi) \triangleq \mathbb{E}_{(\mathcal{O},x_0,g)\sim\mathcal{T}}[\textsc{Time}(\textsc{Plan}(x_0, g, \Psi, \Omega, \Sigma))],$$

where Ω and Σ are learned using Ψ as we described in Sections A.1 and A.3, PLAN is the algorithm described in Section 4, and TIME$(\cdot)$ measures the time that PLAN takes to find a solution[2]. However, we need an objective that can be used to guide a *search* over candidate predicate sets, meaning the objective must be evaluated many times. J_{real} is far too expensive for this, due to two speed bottlenecks: sampler learning, which involves training several neural networks; and the repeated calls to REFINE from within PLAN, which each perform backtracking search to refine an abstract plan. To overcome this intractability, we will use a *surrogate objective* J_{surr} that is cheaper to evaluate than J_{real}, but that approximately preserves the ordering over predicate sets, i.e., $J_{\text{surr}}(\Psi) < J_{\text{surr}}(\Psi') \iff J_{\text{real}}(\Psi) < J_{\text{real}}(\Psi')$.

We propose a surrogate objective that uses the demonstrations $\mathcal{D}$ to *estimate* the time it would take to solve the training tasks under the abstraction induced by a candidate predicate set Ψ, without using samplers or doing refinement. Recalling that $\mathcal{D}$ has one demonstration π^* for each training task $\langle \mathcal{O}, x_0, g \rangle$, the objective is defined as follows:

$$J_{\text{surr}}(\Psi) \triangleq \frac{1}{|\mathcal{D}|} \sum_{(\mathcal{O},x_0,g,\pi^*)\in\mathcal{D}} [\text{ETPT}(x_0, g, \Psi, \Omega, \pi^*)],$$

where ETPT abbreviates Estimate Total Planning Time (Algorithm 2). ETPT uses the candidate predicates and induced operators to perform the first part of bilevel planning: A* search over abstract plans. However, for each generated abstract plan, rather than learning samplers and calling REFINE, we use the available demonstrations to estimate the probability that refinement *would* succeed if we *were* to learn samplers and call REFINE. Since bilevel planning terminates upon the successful refinement of an abstract plan, we can use these probabilities to approximate the total expected planning time. We now describe these steps in detail.

Estimating Refinement Probability ETPT maintains a probability $p_{\text{terminate}}$, initialized to 0 (Line 2), that planning would terminate after each generated abstract plan. To update $p_{\text{terminate}}$ (Lines 5-6), we must estimate both whether PLAN would have terminated *before* this step, and whether PLAN would terminate *on* this step. For the former, we can use $(1 - p_{\text{terminate}})$. For the latter, since PLAN terminates only if REFINE succeeds, we use a helper function ESTIMATEREFINEPROB to approximate the probability of successfully refining the given abstract plan, if we were to learn samplers Σ and then call REFINE. We use the following implementation:

$$\textsc{EstimateRefineProb}(\hat{\pi}, \pi^*) \triangleq (1-\epsilon)\epsilon^{|\textsc{Cost}(\hat{\pi})-\textsc{Cost}(\pi^*)|}.$$

Here, $\epsilon > 0$ is a small constant (10^{-5} in our experiments), and COST$(\cdot)$ is in our case simply the number of actions in the plan, due to unitary costs. The intuition for this geometric distribution is as follows. Since the demonstration π^* is assumed to be near-optimal, an abstract plan $\hat{\pi}$ that is cheaper than π^* should look suspicious; if such a $\hat{\pi}$ were refinable, then the demonstrator would have likely used it to produce a better demonstration. If $\hat{\pi}$ is more expensive than π^*, then even though this abstraction would eventually produce a refinable abstract plan, it may take a long time for the outer loop of the planner, GENABSTRACTPLAN, to get to it (Section 4). We note that this scheme for estimating refinability is surprisingly minimal, in that it needs only the cost of each demonstration rather than its contents.

[2]If no plan can be found (e.g., a task is infeasible under the abstraction), TIME would return a large constant representing a timeout.

Estimating Time To approximate the total planning time, ETPT estimates the time required for each generated abstract plan, conditioned on its successful refinement, and then uses the refinement probabilities to compute the total expectation. The time estimate is maintained in t_{expected}, initialized to 0 (Line 3). To update t_{expected} on each abstract plan (Lines 7-8), we use a helper function ESTIMATETIME, which sums together estimates of the *abstract search time* and of the *refinement time*. Since we are running abstract search, we could exactly measure its time; however, to avoid noise due to CPU speed, we instead use the cumulative number of nodes created by the A* search. To estimate refinement time, recall that REFINE performs a backtracking search, and so over many calls to REFINE, the potentially several that fail will dominate the one or zero that succeed. Therefore, we estimate refinement time as a large constant (10^3 in our experiments) that captures the average cost of an exhaustive backtracking search. Finally, we use a large constant t_{upper} (10^5 in our experiments) to penalize in the case where no abstract plan succeeds (Line 9).

What is the ideal choice for n_{abstract}, the maximum number of abstract plans to consider within ETPT? From an efficiency perspective, $n_{\text{abstract}} = 1$ is ideal, but otherwise, it is not obvious whether to prefer the value of n_{abstract} that will eventually be used with PLAN at evaluation time, or to instead prefer $n_{\text{abstract}} = \infty$. On one hand, we want ETPT to be as much of a mirror image of PLAN as possible; on the other hand, some experimentation we conducted suggests that a larger value of n_{abstract} can smooth the objective landscape, which makes search easier. In practice, it may be advisable to treat n_{abstract} as a hyperparameter.

In summary, our surrogate objective J_{surr} calculates and combines two characteristics of a candidate predicate set Ψ: (1) abstract plan cost "error," i.e., $|\text{COST}(\hat{\pi}) - \text{COST}(\pi^*)|$; and (2) abstract planning time, i.e., number of nodes created during A*. The first feature uses only the costs of the demonstrated plans, while the second feature does not use the demonstrated plans at all. In Appendix A.7, we conduct an empirical analysis to further unpack the contribution of these two features to the overall surrogate objective, finding them to be helpful together but insufficient individually.

5.2 Local Search over Candidate Predicate Sets

With our surrogate objective J_{surr} established, we turn to the question of how to best optimize it. We use a simple hill-climbing search, initialized with $\Psi_0 \leftarrow \Psi_G$, and adding a single new predicate ψ from the pool on each step i:

$$\Psi_{i+1} \leftarrow \operatorname*{argmin}_{\psi \notin \Psi_i} J_{\text{surr}}(\Psi_i \cup \{\psi\}).$$

We repeat until no improvement can be found, and use the last predicate set as our final Ψ. See Figure 2 for an example taken from our experiments in the Blocks environment.

Designing a Grammar of Predicates Designing a grammar of predicates can be difficult, since there is a tradeoff between the expressivity of the grammar and the practicality of searching over it. For our experiments, we found that a simple grammar similar to that of Pasula, Zettlemoyer, and Kaelbling (2007) suffices, which includes single-feature inequalities, logical negation, and universal quantification. See Section A.4 for a full description and Figure 1 and Appendix A.7 for examples.

The costs accumulated over the production rules lead us to a final cost associated with each predicate ψ, denoted $\text{PEN}(\psi)$, where a higher cost represents a predicate with higher complexity. We use the costs to regularize J_{surr} during local search, with a weight small enough to primarily prevent the addition of "neutral" predicates that neither harm nor hurt J_{surr}. The regularization term is $J_{\text{reg}}(\Psi) \triangleq w_{\text{reg}} \sum_{\psi \in \Psi} \text{PEN}(\psi)$, where $w_{\text{reg}} = 10^{-4}$ in our experiments. To generate our candidate predicate set for local search, we enumerate n_{grammar} (200 in experiments) predicates from the grammar, in order of increasing cost.

6 Experiments

Our experiments are designed to answer the following questions: **(Q1)** To what extent do our learned abstractions help both the effectiveness and the efficiency of planning, and how do they compare to abstractions learned using other objective functions? **(Q2)** How do our learned state abstractions compare in performance to manually designed state abstractions? **(Q3)** How data-efficient is learning, with respect to the number of demonstrations? **(Q4)** Do our abstractions vary as we change the planner configuration, and if so, how?

Experimental Setup We evaluate 10 methods across four robotic planning environments. All results are averaged over 10 random seeds. For each seed, we sample a set of 50 *evaluation tasks* that involve more objects and harder goals than were seen at training. Demonstrations are collected by bilevel planning with manually defined abstractions (see Manual method below). Planning is always limited to a 10-second timeout. See Appendix A.6 for additional details.

Environments We now briefly describe the environments, with further details in Appendix A.5. The first three environments were established in prior work by Silver et al. (2021), but in that work, all predicates were manually defined; we use the same predicates in the Manual baseline.

- **PickPlace1D.** A robot must pick blocks and place them onto target regions along a table surface. All pick and place poses are in a 1D line. Evaluation tasks require 1-4 actions to solve.
- **Blocks.** A robot in 3D must interact with blocks on a table to assemble them into towers. This is a robotic version of the classic blocks world domain. Evaluation tasks require 2-20 actions to solve.
- **Painting.** A robot in 3D must pick, wash, dry, paint, and place widgets into either a box or a shelf. Evaluation tasks require 11-25 actions to solve.
- **Tools.** A robot operating on a 2D table surface must assemble contraptions with screws, nails, and bolts, using a provided set of screwdrivers, hammers, and wrenches respectively. This environment has physical constraints that cannot be modeled by our predicate grammar. Evaluation tasks require 7-20 actions to solve.

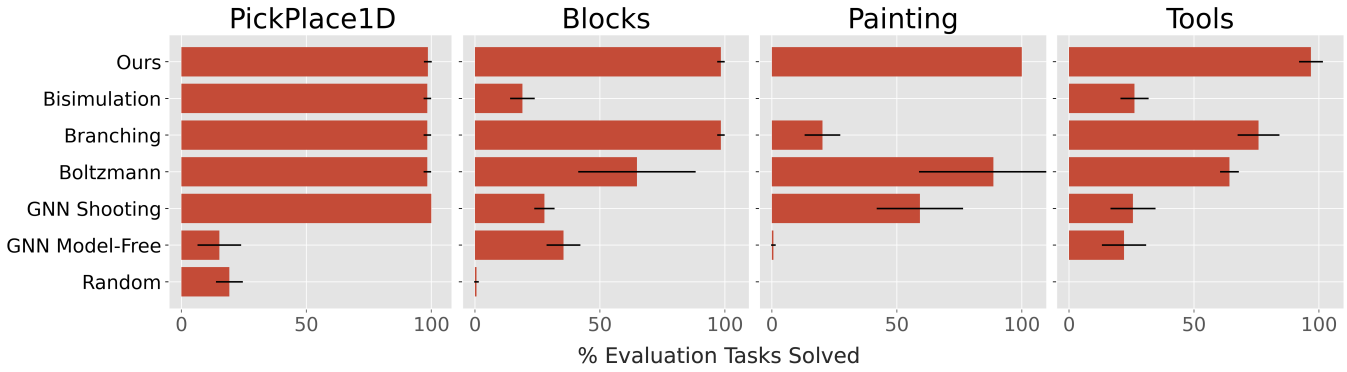


Figure 3: **Ours versus baselines.** Percentage of 50 evaluation tasks solved under a 10-second timeout, for all four environments. All results are averaged over 10 seeds. Black bars denote standard deviations. Learning times and additional metrics are reported in Appendix A.7.

	Ours			**Manual**			**Down Eval**			**No Invent**		
Environment	Succ	Node	Time	Succ	Node	Time	Succ	Node	Time	Succ	Node	Time
PickPlace1D	98.6	4.8	0.006	98.4	6.5	0.045	98.6	4.8	0.008	39.6	14.1	1.369
Blocks	98.4	2949	0.296	98.6	2941	0.251	98.2	2949	0.318	3.2	427.7	1.235
Painting	100.0	501.8	0.470	99.6	2608	0.464	98.8	489.0	0.208	0.0	–	–
Tools	96.8	1897	0.457	100.0	4771	0.491	42.8	152.5	0.060	0.0	–	–

Table 1: **Ours versus Manual and ablations.** Percentage of 50 evaluation tasks solved under a 10-second timeout (Succ), number of nodes created during GENABSTRACTPLAN (Node), and wall-clock planning time in seconds (Time). All results are averaged over 10 seeds. The Node and Time columns average over *solved tasks only*. Standard deviations are provided in Appendix A.7.

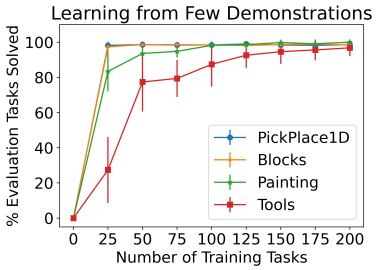
Methods We evaluate our method, six baselines, a manually designed state abstraction, and two ablations. Note that the Bisimulation, Branching, Boltzmann, and Manual baselines differ from Ours only in predicate learning.

- **Ours.** Our main approach.
- **Bisimulation.** A baseline that learns abstractions by approximately optimizing the *bisimulation criteria* (Givan, Dean, and Greig 2003), as in prior work (Curtis et al. 2021). Specifically, this baseline learns abstractions that minimize the number of transitions in the demonstrations where the abstract transition model F is applicable but makes a misprediction about the next abstract state. Note that because goal predicates are given, goal distinguishability is satisfied under any abstraction.
- **Branching.** A baseline that learns abstractions by optimizing the *branching factor* of planning. Specifically, this baseline learns predicates that minimize the number of applicable operators over demonstration states.
- **Boltzmann.** A baseline that assumes the demonstrator is acting *noisily rationally* under (unknown) optimal abstractions (Baker, Saxe, and Tenenbaum 2009). For any candidate abstraction, we compute the likelihood of the demonstration under a Boltzmann policy using the planning heuristic as a surrogate for the true cost-to-go.
- **GNN Shooting.** A baseline that trains a graph neural network (Battaglia et al. 2018) policy. This GNN takes in the current state x, abstract state s, and goal g. It outputs an action a, via a one-hot vector over $\mathcal{C}$ corresponding to which controller to execute, one-hot vectors over all objects at each discrete argument position, and a vector of continuous arguments. We train the GNN using behavior cloning on the data $\mathcal{D}$. At evaluation time, we sample trajectories by treating the outputted continuous arguments as the mean of a Gaussian with fixed variance. We use the transition model f to check if the goal is achieved, and repeat until the planning timeout is reached.
- **GNN Model-Free.** A baseline that uses the same GNN, but directly executes the policy instead of shooting.
- **Random.** A baseline that simply executes a random controller with random arguments on each step. No learning.
- **Manual.** An oracle approach that plans with manually designed predicates for each environment.
- **Down Eval.** An ablation of Ours that uses $n_{\text{abstract}} = 1$ during evaluation only, in PLAN (Algorithm 1).
- **No Invent.** An ablation of Ours that uses $\Psi = \Psi_G$, i.e., only goal predicates are used for the state abstraction.

Results and Discussion We provide real examples of learned predicates and operators for all environments in Appendix A.7. Figure 3 shows that our method solves many more held-out tasks within the timeout than the baselines. A major reason for this performance gap is that our surrogate objective J_{surr} explicitly approximates the efficiency of planning. The lackluster performance of the bisimulation baseline is especially notable because of its prevalence in the literature (Pasula, Zettlemoyer, and Kaelbling 2007; Jetchev, Lang, and Toussaint 2013; Bonet and Geffner 2019; Curtis et al. 2021). We examined its failure modes more closely and found that it consistently selects good predicates, but not *enough* of them. This is because requiring the operators to be a perfect predictive model in the abstract spaces is often not enough to ensure good planning performance. For example, in the Blocks environment, the goal predi-

cates together with the predicate `Holding(?block)` are enough to satisfy bisimulation on our data, while other predicates like `Clear(?block)` and `HandEmpty()` are useful from a planning perspective. Examining the GNN baselines, we see that while shooting is beneficial versus using the GNN model-free, the performance is generally far worse than Ours. Additional experimentation we conducted suggests that the GNN gets better with around an order of magnitude more data.

The figure on the right illustrates the data efficiency of Ours. Each point shows a mean over 10 seeds, with standard deviations shown as vertical bars. We often obtain very good evaluation performance within just 50 demonstrations.



In Table 1, the results for No Invent show that, as expected, the goal predicates alone are completely insufficient for most tasks. Comparing Ours to Down Eval shows that assuming downward refinability at evaluation time works for PickPlace1D, Blocks, and Painting, but not for Tools. We also find that the learned predicates (Ours) are on par with, and sometimes *better than*, hand-designed predicates (Manual). For instance, consider PickPlace1D, where the learned predicates are 7.5x better. The manually designed predicates were `Held(?block)` and `HandEmpty()`, and the always-given goal predicate `Covers(?block, ?target)`. In addition to inventing two predicates that are equivalent to `Held` and `HandEmpty`, Ours invented two more: `P3(?block)` $\triangleq$ $\forall$`?t` . $\neg$`Covers(?block, ?t)`, and `P4(?target)` $\triangleq$ $\forall$`?b` . $\neg$`Covers(?b, ?target)`. Intuitively, `P3` means "the given block is not on any target," while `P4` means "the given target is clear." `P3` gets used in an operator precondition for picking, which reduces the branching factor of abstract search. This precondition is sensible because there is no use in moving a block once it is already on its target. `P4` prevents considering non-refinable abstract plans that "park" objects on targets that must be covered by other objects.

In Appendix A.8, we describe an additional experiment where we vary the AI planning heuristic used in abstract search. We analyze a case in Blocks where variation in the invented predicates appears inconsequential upon initial inspection, but actually has substantial impact on planning efficiency. This result underscores the benefit of using a surrogate objective for predicate invention that is sensitive to downstream planning efficiency.

7 Related Work

Our work continues a long line of research on learning state abstractions for decision-making (Bertsekas, Castanon et al. 1988; Andre and Russell 2002; Jong and Stone 2005; Li, Walsh, and Littman 2006; Abel, Hershkowitz, and Littman 2017; Zhang et al. 2020). Most relevant are works that learn symbolic abstractions compatible with AI planners (Lang, Toussaint, and Kersting 2012; Jetchev, Lang, and Toussaint 2013; Ugur and Piater 2015; Asai and Fukunaga 2018; Bonet and Geffner 2019; Asai and Muise 2020; Ahmetoglu et al. 2020; Umili et al. 2021). Our work is particularly influenced by Pasula, Zettlemoyer, and Kaelbling (2007), who use search through a concept language to invent symbolic state and action abstractions, and Konidaris, Kaelbling, and Lozano-Perez (2018), who discover symbolic abstractions by leveraging the initiation and termination sets of options that satisfy an abstract subgoal property. The objectives used in these prior works are based on variations of auto-encoding, prediction error, or bisimulation, which stem from the perspective that the abstractions should *replace* planning in the original transition space, rather than *guide* it.

Recent works have also considered learning abstractions for multi-level planning, like those in the task and motion planning (TAMP) (Gravot, Cambon, and Alami 2005; Garrett et al. 2021) and hierarchical planning (Bercher, Alford, and Höller 2019) literature. Some of these efforts consider learning symbolic action abstractions (Zhuo et al. 2009; Nguyen et al. 2017; Silver et al. 2021; Aineto, Jiménez, and Onaindia 2022) or refinement strategies (Chitnis et al. 2016; Mandalika et al. 2019; Chitnis, Kaelbling, and Lozano-Pérez 2019; Wang et al. 2021; Chitnis et al. 2022; Ortiz-Haro et al. 2022); our operator and sampler learning methods take inspiration from these prior works. Recent efforts by Loula et al. (2019) and Curtis et al. (2021) consider learning both state and action abstractions for TAMP, like we do (Loula et al. 2019, 2020; Curtis et al. 2021). The main distinguishing feature of our work is that our abstraction learning framework explicitly optimizes an objective that considers downstream planning efficiency.

8 Conclusion and Future Work

In this paper, we have described a method for learning predicates that are explicitly optimized for efficient bilevel planning. Key areas for future work include (1) learning better abstractions from even fewer demonstrations by performing active learning to gather more data online; (2) expanding the expressivity of the grammar to learn more sophisticated predicates; (3) applying these ideas to partially observed planning problems; and (4) learning the controllers that we assumed given in this work.

For (1), we hope to investigate how relational exploration algorithms (Chitnis et al. 2020) might be useful as a mechanism for an agent to decide what actions to execute, toward the goal of building better state and action abstractions. For (2), we can take inspiration from program synthesis, especially methods that can learn programs with continuous parameters (Ellis et al. 2020). For (3) we could draw insights from recent advances in task and motion planning in the partially observed setting (Garrett et al. 2020). Finally, for (4), we recently proposed a method for learning controllers from demonstrations assuming known predicates (Silver et al. 2022). If we can remove the latter assumption, we will have a complete pipeline for learning predicates, operators, samplers, and controllers for bilevel planning.

Acknowledgements

We gratefully acknowledge support from NSF grant 2214177; from AFOSR grant FA9550-22-1-0249; from ONR MURI grant N00014-22-1-2740; from the MIT-IBM Watson Lab; and from the MIT Quest for Intelligence. Tom, Nishanth and Willie are supported by NSF Graduate Research Fellowships. We thank Michael Katz, Christian Muise, Aidan Curtis, Jiayuan Mao, Zhutian Yang, and Amber Li for helpful comments on an earlier draft.

References

Abel, D.; Hershkowitz, D. E.; and Littman, M. L. 2017. Near optimal behavior via approximate state abstraction. *arXiv preprint arXiv:1701.04113*.

Ahmetoglu, A.; Seker, M. Y.; Piater, J.; Oztop, E.; and Ugur, E. 2020. Deepsym: Deep symbol generation and rule learning from unsupervised continuous robot interaction for planning. *arXiv preprint arXiv:2012.02532*.

Aineto, D.; Jiménez, S.; and Onaindia, E. 2022. A Comprehensive Framework for Learning Declarative Action Models. *Journal of Artificial Intelligence Research*, 74: 1091–1123.

Alkhazraji, Y.; Frorath, M.; Grützner, M.; Helmert, M.; Liebetraut, T.; Mattmüller, R.; Ortlieb, M.; Seipp, J.; Springenberg, T.; Stahl, P.; and Wülfing, J. 2020. Pyperplan.

Andre, D.; and Russell, S. J. 2002. State abstraction for programmable reinforcement learning agents. In *AAAI/IAAI*, 119–125.

Asai, M.; and Fukunaga, A. 2018. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *AAAI*.

Asai, M.; and Muise, C. 2020. Learning neural-symbolic descriptive planning models via cube-space priors: The voyage home (to STRIPS). *arXiv preprint arXiv:2004.12850*.

Bacchus, F. 2001. AIPS 2000 planning competition: The fifth international conference on artificial intelligence planning and scheduling systems. *Ai magazine*, 22(3): 47–47.

Baker, C. L.; Saxe, R.; and Tenenbaum, J. B. 2009. Action understanding as inverse planning. *Cognition*.

Battaglia, P. W.; Hamrick, J. B.; Bapst, V.; Sanchez-Gonzalez, A.; Zambaldi, V.; Malinowski, M.; Tacchetti, A.; Raposo, D.; Santoro, A.; Faulkner, R.; et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning-One Abstract Idea, Many Concrete Realizations. In *IJCAI*, 6267–6275.

Bertsekas, D. P.; Castanon, D. A.; et al. 1988. Adaptive aggregation methods for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*.

Bonet, B.; and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence*, 129(1-2): 5–33.

Bonet, B.; and Geffner, H. 2019. Learning first-order symbolic representations for planning from the structure of the state space. *arXiv preprint arXiv:1909.05546*.

Chitnis, R.; Hadfield-Menell, D.; Gupta, A.; Srivastava, S.; Groshev, E.; Lin, C.; and Abbeel, P. 2016. Guided search for task and motion plans using learned heuristics. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 447–454. IEEE.

Chitnis, R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2019. Learning quickly to plan quickly using modular meta-learning. In *2019 International Conference on Robotics and Automation (ICRA)*, 7865–7871. IEEE.

Chitnis, R.; Silver, T.; Tenenbaum, J.; Kaelbling, L. P.; and Lozano-Pérez, T. 2020. GLIB: Efficient exploration for relational model-based reinforcement learning via goal-literal babbling. *arXiv preprint arXiv:2001.08299*.

Chitnis, R.; Silver, T.; Tenenbaum, J. B.; Lozano-Pérez, T.; and Kaelbling, L. P. 2022. Learning Neuro-Symbolic Relational Transition Models for Bilevel Planning. In *The IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Clevert, D.-A.; Unterthiner, T.; and Hochreiter, S. 2015. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*.

Cropper, A.; and Muggleton, S. H. 2016. Learning Higher-Order Logic Programs through Abstraction and Invention. In *IJCAI*, 1418–1424.

Curtis, A.; Silver, T.; Tenenbaum, J. B.; Lozano-Perez, T.; and Kaelbling, L. P. 2021. Discovering State and Action Abstractions for Generalized Task and Motion Planning. *arXiv preprint arXiv:2109.11082*.

Dantam, N. T.; Kingston, Z. K.; Chaudhuri, S.; and Kavraki, L. E. 2016. Incremental task and motion planning: A constraint-based approach. In *Robotics: Science and systems*, volume 12, 00052. Ann Arbor, MI, USA.

Ellis, K.; Wong, C.; Nye, M.; Sable-Meyer, M.; Cary, L.; Morales, L.; Hewitt, L.; Solar-Lezama, A.; and Tenenbaum, J. B. 2020. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*.

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

Garrett, C. R.; Chitnis, R.; Holladay, R.; Kim, B.; Silver, T.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021. Integrated task and motion planning. *Annual review of control, robotics, and autonomous systems*, 4: 265–293.

Garrett, C. R.; Paxton, C.; Lozano-Pérez, T.; Kaelbling, L. P.; and Fox, D. 2020. Online replanning in belief space for partially observable task and motion problems. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 5678–5684. IEEE.

Givan, R.; Dean, T.; and Greig, M. 2003. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence*, 147(1-2): 163–223.

Gravot, F.; Cambon, S.; and Alami, R. 2005. aSyMov: a planner that deals with intricate symbolic and geometric problems. In *Robotics Research. The Eleventh International Symposium*, 100–110.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, critical paths and abstractions: what's the difference anyway? In *Nineteenth International Conference on Automated Planning and Scheduling*.

Jetchev, N.; Lang, T.; and Toussaint, M. 2013. Learning grounded relational symbols from continuous data for abstract reasoning. In *Proceedings of the 2013 ICRA Workshop on Autonomous Learning*.

Jong, N. K.; and Stone, P. 2005. State Abstraction Discovery from Irrelevant State Variables. In *IJCAI*.

Katz, M.; Sohrabi, S.; Udrea, O.; and Winterer, D. 2018. A Novel Iterative Approach to Top-k Planning. In *Proceedings of the Twenty-Eigth International Conference on Automated Planning and Scheduling (ICAPS 2018)*. AAAI Press.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Konidaris, G.; and Barto, A. 2009. Efficient skill learning using abstraction selection. In *Twenty-First International Joint Conference on Artificial Intelligence*.

Konidaris, G.; Kaelbling, L. P.; and Lozano-Perez, T. 2018. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, 61: 215–289.

Lang, T.; Toussaint, M.; and Kersting, K. 2012. Exploration in relational domains for model-based reinforcement learning. *The Journal of Machine Learning Research*, 13(1): 3725–3768.

Lavrac, N.; and Dzeroski, S. 1994. Inductive Logic Programming. In *Logic Programming Workshop*, 146–160.

Li, L.; Walsh, T. J.; and Littman, M. L. 2006. Towards a Unified Theory of State Abstraction for MDPs. *ISAIM*.

Loula, J.; Allen, K.; Silver, T.; and Tenenbaum, J. 2020. Learning constraint-based planning models from demonstrations. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 5410–5416. IEEE.

Loula, J.; Silver, T.; Allen, K. R.; and Tenenbaum, J. 2019. Discovering a symbolic planning language from continuous experience. In *Annual Meeting of the Cognitive Science Society (CogSci)*, 2193.

Mandalika, A.; Choudhury, S.; Salzman, O.; and Srinivasa, S. 2019. Generalized lazy search for robot motion planning: Interleaving search and edge evaluation via event-based toggles. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, 745–753.

Marthi, B.; Russell, S. J.; and Wolfe, J. A. 2007. Angelic Semantics for High-Level Actions. In *ICAPS*, 232–239.

Menon, A.; Tamuz, O.; Gulwani, S.; Lampson, B.; and Kalai, A. 2013. A machine learning framework for programming by example. In *International Conference on Machine Learning*, 187–195. PMLR.

Nguyen, C.; Reifsnyder, N.; Gopalakrishnan, S.; and Munoz-Avila, H. 2017. Automated learning of hierarchical task networks for controlling minecraft agents. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, 226–231. IEEE.

Ortiz-Haro, J.; Ha, J.-S.; Driess, D.; and Toussaint, M. 2022. Structured deep generative models for sampling on constraint manifolds in sequential manipulation. In *Conference on Robot Learning*, 213–223. PMLR.

Pasula, H. M.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2007. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29: 309–352.

Ramírez, M.; and Geffner, H. 2010. Probabilistic plan recognition using off-the-shelf classical planners. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*.

Ren, T.; Chalvatzaki, G.; and Peters, J. 2021. Extended Tree Search for Robot Task and Motion Planning. *arXiv preprint arXiv:2103.05456*.

Silver, T.; Athalye, A.; Tenenbaum, J. B.; Lozano-Perez, T.; and Kaelbling, L. P. 2022. Learning Neuro-Symbolic Skills for Bilevel Planning. In *Conference on Robot Learning*.

Silver, T.; Chitnis, R.; Tenenbaum, J.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021. Learning symbolic operators for task and motion planning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3182–3189. IEEE.

Srivastava, S.; Fang, E.; Riano, L.; Chitnis, R.; Russell, S.; and Abbeel, P. 2014. Combined task and motion planning through an extensible planner-independent interface layer. In *2014 IEEE international conference on robotics and automation (ICRA)*, 639–646. IEEE.

Stahl, I. 1993. Predicate invention in ILP—an overview. In *European Conference on Machine Learning*, 311–322.

Ugur, E.; and Piater, J. 2015. Bottom-up learning of object categories, action effects and logical rules: From continuous manipulative exploration to symbolic planning. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2627–2633. IEEE.

Umili, E.; Antonioni, E.; Riccio, F.; Capobianco, R.; Nardi, D.; and De Giacomo, G. 2021. Learning a Symbolic Planning Domain through the Interaction with Continuous Environments. *ICAPS PRL Workshop*.

Wang, Z.; Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021. Learning compositional models of robot skills for task and motion planning. *The International Journal of Robotics Research*, 40(6-7): 866–894.

Zhang, A.; McAllister, R.; Calandra, R.; Gal, Y.; and Levine, S. 2020. Learning invariant representations for reinforcement learning without reconstruction. *arXiv preprint arXiv:2006.10742*.

Zhi-Xuan, T.; Mann, J.; Silver, T.; Tenenbaum, J.; and Mansinghka, V. 2020. Online bayesian goal inference for boundedly rational planning agents. *Advances in Neural Information Processing Systems*, 33: 19238–19250.

Zhuo, H. H.; Hu, D. H.; Hogg, C.; Yang, Q.; and Munoz-Avila, H. 2009. Learning HTN method preconditions and action models from partial observations. In *Twenty-First International Joint Conference on Artificial Intelligence*.

A Appendix

A.1 Operator Learning

Here we describe how to learn operators Ω, assuming that the full set of predicates Ψ is already learned (Chitnis et al. 2022). This method makes two restrictions on the representation that together lead to very efficient operator learning (linear time in the number of transitions in $\mathcal{D}$). First, for each CON and each possible effect set pair $(\text{EFF}^+, \text{EFF}^-)$, there is at most one operator with that $(\text{CON}, \text{EFF}^+, \text{EFF}^-)$. This restriction makes it impossible to learn multiple operators with different preconditions for the same controller and effect sets. Second, each parameter in PAR must appear in PAR_{CON}, EFF^+, or EFF^-. This restriction prevents modeling "indirect effects," where some object impacts the execution of a controller without its own state being changed. Though these two restrictions are limiting, we are willing to accept them because predicate invention can compensate. For example, an invented predicate can quantify out an object that does not appear in the controller or the effects, to capture indirect effects.

With these restrictions established, we learn operators from our demonstrations $\mathcal{D}$ and predicates Ψ in three steps. Note that each demonstration can be expressed as a sequence of transitions $\{(x, a, x')\}$, with $x, x' \in \mathcal{X}$ and $a \in \mathcal{A}$. First, we use Ψ to ABSTRACT all states x, x' in the demonstrations $\mathcal{D}$, creating a dataset of transitions $\{(s, a, s')\}$ with $s, s' \in \mathcal{S}_\Psi$. Next, we partition these transitions using the following equivalence relation: $(s_1, a_1, s_1') \equiv (s_2, a_2, s_2')$ if the effects and partially specified controllers *unify*, that is, if there exists a mapping between the objects such that a_1, $(s_1 - s_1')$, and $(s_1' - s_1)$ are equivalent to a_2, $(s_2 - s_2')$, and $(s_2' - s_2)$ respectively. This partitioning step automatically determines the number of operators that will ultimately be learned: each equivalence class will induce one operator. Furthermore, the parameters PAR, controller tuple CON, and effects $(\text{EFF}^+, \text{EFF}^-)$ of the operators can now be established as follows. For each equivalence class, we create PAR by selecting an arbitrary transition (s, a, s') and replacing each object that appears in the controller or effects with a variable of the same type. This further induces a substitution $\delta : \text{PAR} \to \mathcal{O}$ for the objects $\mathcal{O}$ in this transition; the CON, EFF^+, and EFF^- are then created by applying δ to a, $(s' - s)$, and $(s - s')$ respectively. By construction, for all other transitions τ in the same equivalence class, there exists an injective substitution δ_τ under which the controller arguments and effects are equivalent to the newly created CON, EFF^+, and EFF^-. We use these substitutions for the third and final step of operator learning: precondition learning. For this, we perform an intersection over all abstract states in each equivalence class (Bonet and Geffner 2019; Chitnis et al. 2022; Curtis et al. 2021): $\text{PRE} \leftarrow \bigcap_{\tau=(s,\cdot,\cdot)} \delta_\tau^{-1}(s)$, where $\delta_\tau^{-1}(s)$ substitutes all occurrences of the objects in s with the parameters in PAR following an inversion of δ_τ, and discards any atoms involving objects that are not in the image of δ_τ. By this construction, only the parameters in PAR will be involved in PRE, as desired. With PAR, PRE, EFF^+, EFF^-, and CON now established for each equivalence class, we have completed the operators Ω.

Soundness. We note that for any predicates Ψ, the operator learning procedure is *sound* (Konidaris, Kaelbling, and Lozano-Perez 2018) over the data, in the following sense: for each transition $\tau = (x, a, x')$, there exists some $\underline{\omega}$, a learned operator ground with objects in x, such that $F(\text{ABSTRACT}(x, \Psi), \underline{\omega})$ is defined and equals $\text{ABSTRACT}(x', \Psi)$. To see this, recall that τ belongs to an equivalence class, and that this equivalence class was used to learn an operator ω. Now, we show that the desired $\underline{\omega}$ is $\langle \omega, \delta_\tau \rangle$, where δ_τ is the injective parameter-to-object substitution defined above. The $\underline{\text{CON}}$, $\underline{\text{EFF}^+}$, and $\underline{\text{EFF}^-}$ of $\underline{\omega}$ exactly equal those in τ, by construction of the substitution δ_τ. Additionally, because $\underline{\text{PRE}}$ was formed by taking an intersection of abstract states that included $\text{ABSTRACT}(x, \Psi)$, it must be the case that $\underline{\text{PRE}} \subseteq \text{ABSTRACT}(x, \Psi)$, since an intersection must be a subset of every constituent set. By Definition 4, then, the statement is satisfied. A corollary of this soundness property is that our learned abstractions are guaranteed to obey the semantics we defined in Section 3 with respect to the training data.

As a byproduct of operator learning, we have also determined "local" datasets for each operator, with each transition in the respective equivalence class defining an example of the operator's preconditions, controller, and effects. We will use these local datasets and the corresponding substitutions δ_τ during sampler learning Section A.3.

A.2 Operator Learning Extended Example

We conclude our discussion of operator learning with an extended example. We start with a small toy dataset and use it to walk through each of the three steps in the procedure.

Step 1: Generate Dataset. In this example, our demonstrations contain four transitions, which are tuples (x, a, x'). For clarity, we will not write out the task-level states x and x'. Additionally, for the sake of the example, we will assume that in this environment there is only one controller `C`, with no discrete arguments. We abstract these states with the predicate set Ψ includes `Held`, `On`, `IsPurple`, `IsRed`, `IsGreen`, `IsStowable`, and `IsStowed`, which leads to four (s, a, s') tuples:

1. $(\{\texttt{On}(o_1, o_2), \texttt{On}(o_2, o_3), \texttt{IsPurple}(o_1)\}, \texttt{C}(\theta_1), \{\texttt{Held}(o_1), \texttt{On}(o_2, o_3), \texttt{IsPurple}(o_1)\})$
2. $(\{\texttt{On}(o_4, o_5), \texttt{On}(o_5, o_6), \texttt{IsRed}(o_4)\}, \texttt{C}(\theta_2), \{\texttt{Held}(o_4), \texttt{On}(o_5, o_6), \texttt{IsRed}(o_4)\})$
3. $(\{\texttt{Held}(o_1), \texttt{IsStowable}(o_1), \texttt{IsGreen}(o_2)\}, \texttt{C}(\theta_3), \{\texttt{IsStowed}(o_1), \texttt{IsStowable}(o_1), \texttt{IsGreen}(o_2)\})$
4. $(\{\texttt{Held}(o_8), \texttt{IsStowable}(o_8), \texttt{IsGreen}(o_9)\}, \texttt{C}(\theta_4), \{\texttt{IsStowed}(o_8), \texttt{IsStowable}(o_8), \texttt{IsGreen}(o_9)\})$

Intuitively, the first and second transitions might occur when picking up an object (o_1 or o_4 respectively), while the third and fourth might occur when stowing an object (o_1 or o_8 respectively). We begin by noting that we can ignore the continuous parameters θ_i of `C`, since they do not matter for operator learning (they would be used in sampler learning).

Step 2: Produce Equivalence Classes. Recall that two transitions are in the same equivalence class if there exists a mapping between objects such that the controller, controller discrete arguments, and effects are equivalent. Since we only have one controller `C` with no discrete arguments in this example, we must only check for effect equivalence. The first transition has effects $(\text{EFF}^+, \text{EFF}^-) = (\{\texttt{Held}(o_1)\}, \{\texttt{On}(o_1, o_2)\})$, while the second has effects $(\text{EFF}^+, \text{EFF}^-) = (\{\texttt{Held}(o_4)\}, \{\texttt{On}(o_4, o_5)\})$. These can be unified with the mapping $\{o_1 \leftrightarrow o_4, o_2 \leftrightarrow o_5\}$. Similarly, the third transition has effects $(\text{EFF}^+, \text{EFF}^-) = (\{\texttt{IsStowed}(o_1)\}, \{\texttt{Held}(o_1)\})$, while the fourth has effects $(\text{EFF}^+, \text{EFF}^-) = (\{\texttt{IsStowed}(o_8)\}, \{\texttt{Held}(o_1)\})$. These can be unified with the mapping $\{o_1 \leftrightarrow o_8\}$.

Note that in this unification procedure, the atoms which were unchanged, such as $\texttt{IsPurple}(o_1)$, do not play a role. Furthermore, the fact that the objects are the same between transitions 1 and 3 is unimportant, because these transitions belong to different equivalence classes.

Selecting an arbitrary transition from each equivalence class and substituting objects with variables, we get the following:

- Equivalence class 1:
  - PAR: [?x, ?y]
  - EFF^+: {Held(?x)}
  - EFF^-: {On(?x, ?y)}
  - CON: ⟨ C, [] ⟩
  - Transitions contained: 1 and 2
  - δ_1 (substitution for transition 1): $\{\texttt{?x} \to o_1, \texttt{?y} \to o_2\}$
  - δ_2 (substitution for transition 2): $\{\texttt{?x} \to o_4, \texttt{?y} \to o_5\}$
- Equivalence class 2:
  - PAR: [?z]
  - EFF^+: {IsStowed(?z)}
  - EFF^-: {Held(?z)}
  - CON: ⟨ C, [] ⟩
  - Transitions contained: 3 and 4
  - δ_3 (substitution for transition 3): $\{\texttt{?z} \to o_1\}$
  - δ_4 (substitution for transition 4): $\{\texttt{?z} \to o_8\}$

Note that the parameter list PAR for each equivalence class contains all parameters that appear in PAR_{CON}, EFF^+, or EFF^-.

Step 3: Learn Operator Preconditions. We now have all the ingredients of the operators except for their preconditions. For each transition in each equivalence class, we first discard any atom from the abstract state s which involves objects *not* in the image of that transition's substitution δ. For instance, the first transition has $\delta_1 = \{\texttt{?x} \to o_1, \texttt{?y} \to o_2\}$. The image is $\{o_1, o_2\}$, which excludes o_3. This means that the atom $\texttt{On}(o_2, o_3)$ is discarded from s.

After discarding atoms appropriately, we end up with these abstract states for each transition:

1. $\{\texttt{On}(o_1, o_2), \texttt{IsPurple}(o_1)\}$
2. $\{\texttt{On}(o_4, o_5), \texttt{IsRed}(o_4)\}$
3. $\{\texttt{Held}(o_1), \texttt{IsStowable}(o_1)\}$
4. $\{\texttt{Held}(o_8), \texttt{IsStowable}(o_8)\}$

Now, the preconditions for each equivalence class are obtained by applying each δ_i^{-1} to these abstract states and taking intersections. This produces the final operator set Ω, which does not contain any extraneous atoms related to object color:

- Operator 1 (from equivalence class 1):
  - PAR: [?x, ?y]
  - PRE: {On(?x, ?y)}
  - EFF^+: {Held(?x)}
  - EFF^-: {On(?x, ?y)}
  - CON: ⟨ C, [] ⟩
- Operator 2 (from equivalence class 2):
  - PAR: [?z]
  - PRE: {Held(?z), IsStowable(?z)}
  - EFF^+: {IsStowed(?z)}
  - EFF^-: {Held(?z)}
  - CON: ⟨ C, [] ⟩

A.3 Learning Samplers

The role of a sampler $\sigma \in \Sigma$ is to *refine* its associated operator ω, suggesting continuous parameters of actions that will transition the environment from a state where the preconditions hold to a state where the effects follow. Recall that a sampler $\sigma : \mathcal{X} \times \mathcal{O}^{|\text{PAR}|} \to \Delta(\Theta)$ defines a conditional distribution $P(\theta \mid x, o_1, \ldots, o_k)$, where θ are continuous parameters for the controller C in ω, and $(o_1, \ldots, o_k)$ represent a set of objects that could be used to ground ω, with $|\text{PAR}| = k$. Using the same demonstration dataset $\mathcal{D}$, we learn samplers of the following form, one per operator:

$$\sigma(x, o_1, \ldots, o_k) = r_\sigma(x[o_1] \oplus \cdots \oplus x[o_k]),$$

where $x[o]$ denotes the feature vector for o in x, the $\oplus$ denotes concatenation, and r_σ is the model to be learned.

To learn samplers, we use the local datasets created during operator learning (Section A.1), to create datasets for supervised sampler learning, with one dataset per sampler. Consider any (non-abstract) transition $\tau = (x, a, \cdot)$ in the equivalence class associated with an operator ω. To create a datapoint for the associated sampler, we can reuse the substitution δ_τ found during operator learning to create an input vector $x[\delta_\tau(v_1)] \oplus \cdots \oplus x[\delta_\tau(v_k)]$, where $(v_1, \ldots, v_k) = \text{PAR}$. The corresponding output for supervised learning is the continuous parameter vector θ in the action a.

With these datasets created, one could use any method for multidimensional distributional regression to learn each r_σ. In this work, we learn two neural networks to parameterize each sampler. The first neural network takes in $x[o_1] \oplus \cdots \oplus x[o_k]$ and regresses to the mean and covariance matrix of a Gaussian distribution over θ; here, we are assuming that the desired distribution has nonzero measure, but the covariances can be arbitrarily small in practice. This neural network is a sampler in its own right, but its expressive power is limited, e.g., to unimodal distributions. To improve representational capacity, we learn a second neural network

that takes in $x[o_1] \oplus \cdots \oplus x[o_k]$ *and* θ, and returns true or false. This classifier is then used to rejection sample from the first network. To create negative examples, we use all transitions τ' such that the controller in τ' matches that in CON, but the effects in τ' are different from $(\text{EFF}^+, \text{EFF}^-)$.

A.4 Predicate Grammar Details

Here we detail the grammar over predicate candidates used in our experiments. Note that the grammar is the same for all environments (up to the object types and goal predicates).

- The base grammar includes two kinds of predicates: all the goal predicates Ψ_G, and *single-feature inequality classifiers*. These inequality classifiers are less-than-or-equal-to expressions that compare a constant against an individual feature dimension from $\{1, \ldots, d(\lambda)\}$, for some object type $\lambda \in \Lambda$. For the constant, we consider an infinite stream of numbers in the pattern $0.5, 0.25, 0.75, 0.125, 0.375, 0.625, 0.875, \ldots$, which represent *normalized* values of the feature, based on the range of values it takes on across all states in the dataset $\mathcal{D}$. We use this pattern because we want our grammar to describe an infinite stream of classifiers, starting from the median values in $\mathcal{D}$. As an example, a type `block` might have a feature dimension corresponding to its `size`, and a classifier could be `block.size` $\leq$ `0.5`. All goal predicates have cost 0. All single-feature inequality classifiers have cost computed based on the normalized constant, with cost 0 for constant 0.5, cost 1 for constants 0.25 and 0.75, cost 2 for constants 0.125, 0.375, 0.625, 0.875, etc.
- We include all negations of predicates in the base grammar. Negating a predicate adds a cost of 1.
- We include two types of universally quantified predicates over the predicates thus far: (1) quantifying over all variables, and (2) quantifying over all but one variable. An example of the first is `P()` = $\forall$ `?x, ?y . On(?x, ?y)`, while an example of the second is `P(?y)` = $\forall$ `?x . On(?x, ?y)`. Universally quantifying adds a cost of 1.
- We include all negations of universally quantified predicates. Negating a predicate adds a cost of 1.
- Following prior work (Curtis et al. 2021), we prune out candidate predicates if they are equivalent to any previously enumerated predicate, in terms of all groundings that hold in every state in the dataset $\mathcal{D}$. Finally, we discard the goal predicates Ψ_G from the grammar, since they are included in every candidate predicate set Ψ of our search already.

A.5 Additional Environment Details

- **PickPlace1D.** In this toy environment, a robot must pick blocks and place them onto target regions along a table surface. All pick and place poses are in a 1D line. The three object types are block, target, and robot. Blocks and targets have two features for their pose and width, and robots have one feature for the gripper joint state. The block widths are larger than the target widths, and the goal requires each block to be placed so that it completely covers the respective target region, so $\Psi_G = \{\texttt{Covers}\}$, where `Covers` is an arity-2 predicate. There is only one controller, `PickPlace`, with no discrete arguments; its Θ is a single real number denoting the location to perform either a pick or a place, depending on the current state of the robot's gripper. Each action updates the state of at most one block, based on whether any is in a small radius from the continuous parameter θ. Both training tasks and evaluation tasks involve 2 blocks, 2 targets, and 1 robot. In each task, with 75% probability the robot starts out holding a random block; otherwise, both blocks start out on the table. Evaluation tasks require 1-4 actions to solve. This environment was established by Silver et al. (2021), but that work involved manually defined state abstractions, which we do not provide in this paper.
- **Blocks.** In this environment, a robot in 3D must interact with blocks on a table to assemble them into towers. This is a robotics adaptation of the blocks world domain in AI planning. The two object types are block and robot. Blocks have four features: an x/y/z pose and a bit for whether it is currently grasped. Robots have four features: x/y/z end effector pose and the (symmetric) value of the finger joints. The goals involve assembling towers, so $\Psi_G = \{\texttt{On}, \texttt{OnTable}\}$, where the former has arity 2 and describes one block being on top of another, while the latter has arity 1. There are three controllers: `Pick`, `Stack`, and `PutOnTable`. `Pick` is parameterized by a robot and a block to pick up. `Stack` is parameterized by a robot and a block to stack the currently held one onto. `PutOnTable` is parameterized by a robot and a 2D place pose representing normalized coordinates on the table surface at which to place the currently held block. Training tasks involve 3 or 4 blocks, while evaluation tasks involve 5 or 6 blocks; all tasks have 1 robot. In all tasks, all blocks start off in collision-free poses on the table. Evaluation tasks require 2-20 actions to solve. This environment was established by Silver et al. (2021), but that work involved manually defined state abstractions, which we do not provide in this paper.
- **Painting.** In this challenging environment, a robot in 3D must pick, wash, dry, paint, and place widgets into either a box or a shelf, as specified by the goal. The five object types are widget, box, shelf, box lid, and robot. Widgets have eight features: an x/y/z pose, a dirtiness level (requiring washing), a wetness level (requiring drying), a color, a bit for whether it is currently grasped, and the 1D gripper rotation with which it is grasped if so. Boxes and shelves have one feature for their color. Box lids have one feature for whether or not they are open. Robots have one feature for the gripper joint state. The goals involve painting the widgets to be the same color as either a box or a shelf, and then placing each widget into the appropriate one, so $\Psi_G = \{\texttt{InBox}, \texttt{InShelf}, \texttt{IsBoxColor}, \texttt{IsShelfColor}\}$, all of which have arity 2 (a widget, and either a box or a shelf). There are two physical constraints in this environment: (1) placing into a box can only succeed if the robot is top-grasping a widget, while placing into a shelf can only succeed if the robot is side-grasping

it; (2) a box can only be placed into if its respective lid is open. There are six controllers: `Pick`, `Wash`, `Dry`, `Paint`, `Place`, and `OpenLid`. All six are discretely parameterized by a robot argument; `Pick` is additionally parameterized by a widget to pick up, and `OpenLid` by a lid to open. `Pick` has 4 continuous parameters: a 3D grasp pose delta from that widget's center of mass, and a gripper rotation. `Wash`, `Dry`, and `Paint` have 1 continuous parameter each: the amount of washing, the amount of drying, and the desired new color, respectively. `Place` has 3 continuous parameters: a 3D place pose corresponding to where the currently held widget should be placed. Training tasks involve 2 or 3 widgets, while evaluation tasks involve 3 or 4 blocks; all tasks have 1 box, 1 shelf, and 1 robot. In each task, with 50% probability the robot starts out holding a random widget; otherwise, all widgets start out on the table. Also, in each task, with 30% probability the box lid starts out open. Evaluation tasks require 11-25 actions to solve. This environment was established by Silver et al. (2021), but that work involved manually defined state abstractions, which we do not provide in this paper.

- **Tools.** In this challenging environment, a robot operating on a 2D table surface must assemble contraptions by fastening screws, nails, and bolts, using a provided set of screwdrivers, hammers, and wrenches respectively. This environment has physical constraints outside the scope of our predicate grammar, and therefore tests the learner's ability to be robust to an insurmountable lack of downward refinability. The eight object types are contraption, screw, nail, bolt, screwdriver, hammer, wrench, and robot. Contraptions have two features: an x/y pose. Screws, nails, bolts, and the three tools have five features: an x/y pose, a shape, a size, and a bit indicating whether it is held. Robots have one feature for the gripper joint state. The goals involve fastening the screws, nails, and bolts onto target contraptions, so Ψ_G includes `ScrewPlaced`, `NailPlaced`, `BoltPlaced`, `ScrewFastened`, `NailFastened`, and `BoltFastened`. The first three have arity 2 (a screw/nail/bolt and which contraption it is placed on); the last three have arity 1. There are three physical constraints in this environment: (1) a screwdriver can only be used to fasten a screw if its shape is close enough to that of the screw; (2) some screws have a shape that does not match any screwdriver's, and so these screws must be fastened by hand; (3) the three tools cannot be picked up if their sizes are too large. There are eleven controllers: `Pick{Screw, Nail, Bolt, Screwdriver, Hammer, Wrench}`, `Place`, `FastenScrewWithScrewdriver`, `FastenScrewByHand`, `FastenNailWithHammer`, and `FastenBoltWithWrench`. All eleven are discretely parameterized by a robot argument; `Pick` controllers are additionally parameterized by an object to pick up, and `Fasten` controllers by a screw/nail/bolt and tool (except `FastenScrewByHand`, which does not have a tool argument). `Place` has 2 continuous parameters: a 2D place pose corresponding to where the currently held object should be placed, which can be either onto the table or onto a contraption (only if the currently held object is not a tool). Training tasks involve 2 screws/nails/bolts and 2 contraptions, while evaluation tasks involve 2 or 3 screws/nails/bolts and 3 contraptions; all tasks have 3 screwdrivers, 2 hammers, 1 wrench, and 1 robot. Evaluation tasks require 7-20 actions to solve.

A.6 Additional Experimental Details

All experiments were conducted on a quad-core Intel Xeon Platinum 8260 processor. All sampler neural networks are fully connected, with two hidden layers of size 32 each, and trained with the Adam optimizer (Kingma and Ba 2014) for 1K epochs using learning rate 1e-3. The regressor networks are trained to predict a mean and covariance matrix of a multivariate Gaussian; this covariance matrix is restricted to be diagonal and PSD with an exponential linear unit (Clevert, Unterthiner, and Hochreiter 2015). For training the classifier networks, we subsample data to ensure a 1:1 balance between positive and negative examples. All AI planning heuristics are implemented using Pyperplan (Alkhazraji et al. 2020); all experiments use the LMCut heuristic unless otherwise specified. The planning parameters are $n_{\text{abstract}} = 1000$ for Tools and 8 for the other environments, and $n_{\text{samples}} = 1$ for Tools and 10 for the other environments.

A.7 Additional Experimental Results

Table 3 provides learning times for all experiments. Tables 4, 5, and 6 report success rates, nodes created, and wall-clock time respectively for all evaluation tasks.

Figure 4 analyzes the two main features used by our surrogate objective function. See caption for further description.

We now go through each of our four environments, providing an example of learned predicates and operators from a single seed randomly chosen among successful ones. We also provide additional statistics for our main method, to supplement the other results we have provided. Note that the evaluation plan length statistics are averaged over both 10 seeds and 50 evaluation tasks per seed, *with standard deviations over seed only*.

PickPlace1D Statistics for our main method, averaged over 10 random seeds (standard deviations parenthesized):

- Average number of predicates in Ψ (both invented and goal predicates): 5.9 (0.54)
- Average number of operators in Ω: 2.1 (0.3)
- Average plan length during evaluation: 2.44 (0.09)

See Figure 5 for example learned predicates and operators for a randomly chosen successful seed.

Blocks Statistics for our main method, averaged over 10 random seeds (standard deviations parenthesized):

- Average number of predicates in Ψ (both invented and goal predicates): 6.0 (0.0)
- Average number of operators in Ω: 4.0 (0.0)

	Ours			Manual		
Heuristic	Succ	Node	Time	Succ	Node	Time
LMCut	98.4	2949	0.296	98.6	2941	0.251
hAdd	98.6	121.6	0.115	97.8	3883	0.235

Table 2: **Varying planning heuristic**. See text for details.

- Average plan length during evaluation: 9.17 (0.69)

See Figure 6 for example learned predicates and operators for a randomly chosen successful seed.

Painting Statistics for our main method, averaged over 10 random seeds (standard deviations parenthesized):

- Average number of predicates in Ψ (both invented and goal predicates): 22.1 (1.45)
- Average number of operators in Ω: 11.2 (0.6)
- Average plan length during evaluation: 14.76 (0.29)

See Figures 7 and 8 for example learned predicates and operators for a randomly chosen successful seed.

Tools Statistics for our main method, averaged over 10 random seeds (standard deviations parenthesized):

- Average number of predicates in Ψ (both invented and goal predicates): 27.4 (4.39)
- Average number of operators in Ω: 17.8 (0.98)
- Average plan length during evaluation: 10.1 (0.12)

See Figures 9 and 10 for example learned predicates and operators for a randomly chosen successful seed.

A.8 Varying the Planner Heuristic

Table 2 shows an additional experiment we conducted where we varied the AI planning heuristic used by the GENABSTRACTPLAN routine of our bilevel planner in the Blocks environment. Recall that our predicate invention method uses GENABSTRACTPLAN as well, so it too is affected by this heuristic change. All numbers show a mean over 10 seeds. Interestingly, while the gap in performance is limited when using LMCut, our system shows a massive improvement (over 30x fewer nodes created) versus Manual when using hAdd. These results are especially surprising because A* with hAdd is generally considered inferior to other heuristic search algorithms.[3] Inspecting the learned abstractions, we find that our approach invents four unary predicates with the intuitive meanings `Holding`, `NothingAbove`, `HandEmpty`, and `NotOnAnyBlock`, to supplement the given goal predicates `On` and `OnTable`. Comparing these to Manual, which has the same predicates and operators as those in the International Planning Competition (IPC) (Bacchus 2001), we see the following differences: `Clear` is omitted[4], and `NothingAbove` and `NotOnAnyBlock` are added.

We observe that `NothingAbove` and `NotOnAnyBlock` are logical transformations of predicates used in the standard IPC representation. This motivated us to run a separate, symbolic-only experiment, where we collected IPC blocks world problems and transformed them to use these learned predicates. We found that using A* and hAdd, planning with our learned representations is much faster than planning with the IPC representations. For example, in the hardest problem packaged with Pyperplan, which contains 17 blocks, planning with our operators requires approximately 30 seconds and 841 node expansions, whereas planning with the standard encoding requires 560 seconds and 17,795 expansions. We also tried Fast Downward (Helmert 2006) (again with A* and hAdd) on a much harder problem from IPC 2000 with 36 blocks. With our learned representations, planning succeeds in 12.5 seconds after approximately 7,000 expansions, whereas under the standard encoding the planner fails to find a plan within a 2 *hour* timeout. Note that all of these results are specific to Blocks, A*, and hAdd, and that is exactly the point: even when using an unconventional combination of search algorithm and heuristic, our *planner-aware* method learns abstractions that optimize the efficiency of the given planner in the given environment.

Why exactly do our learned predicates and operators outperform the standard ones when planning with A* and hAdd? First, we note that it is highly uncommon to use hAdd with A* in practice with hand-defined PDDL representations, because hAdd is inadmissible and suffers greatly from overestimation issues (Bonet and Geffner 2001). Nevertheless, the interesting phenomenon in our work is that our system is able to *learn an abstraction* that copes with the faults of this combination of search algorithm and heuristic. To understand this further, we make the following observations:

- In both cases, the planner must escape from a local minimum with almost every pick operation. For example, in a small problem with 5 blocks where the hand is initially empty, the hAdd values of the states in the plan found are [9, 13, 9, 11, 6, 8, 4, 5, 2, 1, 0] when planning with the standard operators, and [14, 16, 11, 10, 6, 7, 4, 4, 2, 1, 0] when planning with our learned operators. Note the alternation of increasing and decreasing values; the ideal scenario for planning would instead be that these values decrease smoothly.
- In states that follow a pick, the hAdd values consistently *overestimate* the true cost-to-go, in both cases. For example, after the first pick with the standard operators, the hAdd value and true cost-to-go are 13 and 9 respectively; for the learned operators, they are 16 and 9 respectively.
- Here is the main difference: in states that *precede* a pick, the hAdd values from the standard operators sometimes *underestimate* the true cost-to-go. In the example above, the initial state has an hAdd value of 9, but the true cost-to-go is 10. In harder problems, these underestimations occur with higher frequency; for example, in a problem with 20 blocks, there are 8 cases in the plan found where states preceding picks underestimate the true cost-to-go. In contrast, the hAdd values from our *learned* operators do not ever seem to underestimate the true cost-to-go, in the problems that we analyzed.

[3] We also experimented with GBFS instead of A*, and hFF, hSA, and hMax instead of hAdd. A* with hAdd performed best.

[4] In the standard encoding, "clear" means "nothing above and not holding."

Environment	Ours	Bisimulation	Branching	Boltzmann	GNN Sh	GNN MF	Manual	No Invent
PickPlace1D	625 (134)	176 (3)	219 (145)	264 (17)	1951 (85)	1951 (85)	177 (147)	66 (0)
Blocks	10237 (853)	800 (44)	1561 (98)	9798 (1688)	4047 (209)	4047 (209)	102 (5)	84 (2)
Painting	18395 (28153)	872 (380)	2883 (144)	9457 (3421)	9185 (166)	9185 (166)	565 (457)	260 (2)
Tools	18666 (2815)	573 (20)	5524 (747)	9716 (1000)	7362 (197)	7362 (197)	167 (3)	141 (3)

Table 3: **Learning times in seconds for all experiments**. All numbers are means over 10 seeds, with standard deviations in parentheses. For the GNN-based methods, learning time encompasses training the neural networks. For the other methods, learning time encompasses learning predicates, operators, and samplers (i.e., all components of the abstraction). Even though our main method performs well (Ours), this does come at the cost of increased learning time (although the learning is purely offline). Note that the Manual approach only manually specifies a *state* abstraction (predicates); operators and samplers must still be learned, contributing to the non-zero learning time. Thus, comparing Ours and Manual shows that the large majority of learning time in our system is spent on predicate invention.

Environment	Ours	Bisimulation	Branching	Boltzmann	GNN Sh	GNN MF	Random	Manual	Down Eval	No Invent
PickPlace1D	98.6 (1.6)	98.4 (1.5)	98.4 (1.5)	98.4 (1.5)	100.0 (0.0)	15.2 (8.7)	19.2 (5.4)	98.4 (1.5)	98.6 (1.6)	39.6 (4.8)
Blocks	98.4 (1.5)	19.0 (4.9)	98.4 (1.5)	64.8 (23.5)	27.8 (4.0)	35.4 (6.8)	0.6 (0.9)	98.6 (1.6)	98.2 (1.4)	3.2 (2.0)
Painting	100.0 (0.0)	0.0 (0.0)	20.2 (7.1)	88.6 (29.7)	59.2 (17.3)	0.6 (0.9)	0.0 (0.0)	99.6 (0.8)	98.8 (1.8)	0.0 (0.0)
Tools	96.8 (4.7)	26.2 (5.6)	75.8 (8.4)	64.2 (3.7)	25.6 (9.0)	22.0 (8.9)	0.0 (0.0)	100.0 (0.0)	42.8 (10.4)	0.0 (0.0)

Table 4: **Percentage of evaluation tasks solved for all experiments**. All numbers are means over 10 seeds, with 50 evaluation tasks per seed, and with standard deviations in parentheses.

- Furthermore, this underestimation occurs regularly in states that are local minima, immediately preceding states where the heuristic will be an overestimate, so A* struggles greatly. Since nodes are expanded in order of $f = g + h$, that is, cost of the plan so far plus heuristic value, A* will spend time exploring large subtrees rooted at nodes that underestimate true cost-to-go before moving onto the nodes that overestimate it, including those that will ultimately be included in the plan.

For reproducibility, we provide the complete operators used to conduct this experiment. We started from the standard blocks domain PDDL downloaded from the `planning.domains` Github repository, removed the `Clear` predicate, and added the two predicates our system learned, with the intuitive meanings `NothingAbove` and `NotOnAnyBlock`. Problem files were updated accordingly. We ran Fast Downward with the `--search astar(add)` option. Here is the domain file, with changes highlighted in red (deletion) and green (addition):

Environment	Ours	Bisimulation	Branching	Boltzmann	Manual	Down Eval	No Invent
PickPlace1D	4.8 (0.2)	4.7 (0.2)	4.7 (0.2)	5.3 (0.2)	6.5 (0.3)	4.8 (0.2)	14.1 (4.0)
Blocks	2948.5 (1293.2)	46.9 (18.0)	2948.5 (1293.2)	7844.0 (6655.4)	2940.5 (1299.1)	2948.5 (1293.2)	427.7 (83.7)
Painting	501.8 (180.0)	–	876.6 (509.7)	4008.8 (3851.3)	2607.5 (1117.2)	489.0 (190.2)	–
Tools	1897.2 (1404.0)	5247.7 (2560.6)	167.8 (78.4)	909.9 (174.1)	4770.9 (886.8)	152.5 (27.6)	–

Table 5: **Number of nodes created by abstract search during planning in evaluation tasks**. All numbers are means over *solved tasks only* across 10 seeds, with 50 evaluation tasks per seed, and with standard deviations in parentheses.

Environment	Ours	Bisimulation	Branching	Boltzmann	GNN Sh	GNN MF	Random	Manual	Down Eval	No Invent
PickPlace1D	0.006 (0.0)	0.006 (0.0)	0.006 (0.0)	0.005 (0.0)	0.436 (0.1)	0.014 (0.0)	0.004 (0.0)	0.045 (0.0)	0.008 (0.0)	1.369 (0.6)
Blocks	0.296 (0.1)	0.158 (0.1)	0.284 (0.1)	0.954 (0.3)	0.138 (0.1)	0.249 (0.1)	0.006 (0.0)	0.251 (0.1)	0.318 (0.1)	1.235 (1.3)
Painting	0.470 (0.2)	–	4.186 (0.9)	0.600 (0.3)	2.077 (1.2)	0.073 (0.0)	–	0.464 (0.1)	0.208 (0.0)	–
Tools	0.457 (0.3)	0.699 (0.3)	0.109 (0.0)	0.247 (0.0)	0.311 (0.2)	0.043 (0.0)	–	0.491 (0.1)	0.060 (0.0)	–

Table 6: **Total time in seconds for evaluation tasks**. These results encompass planning time (when applicable) and policy or plan inference time (the time taken to produce an action at each step, given the current state). All numbers are means over *solved tasks only* across 10 seeds, with 50 evaluation tasks per seed, and with standard deviations in parentheses.

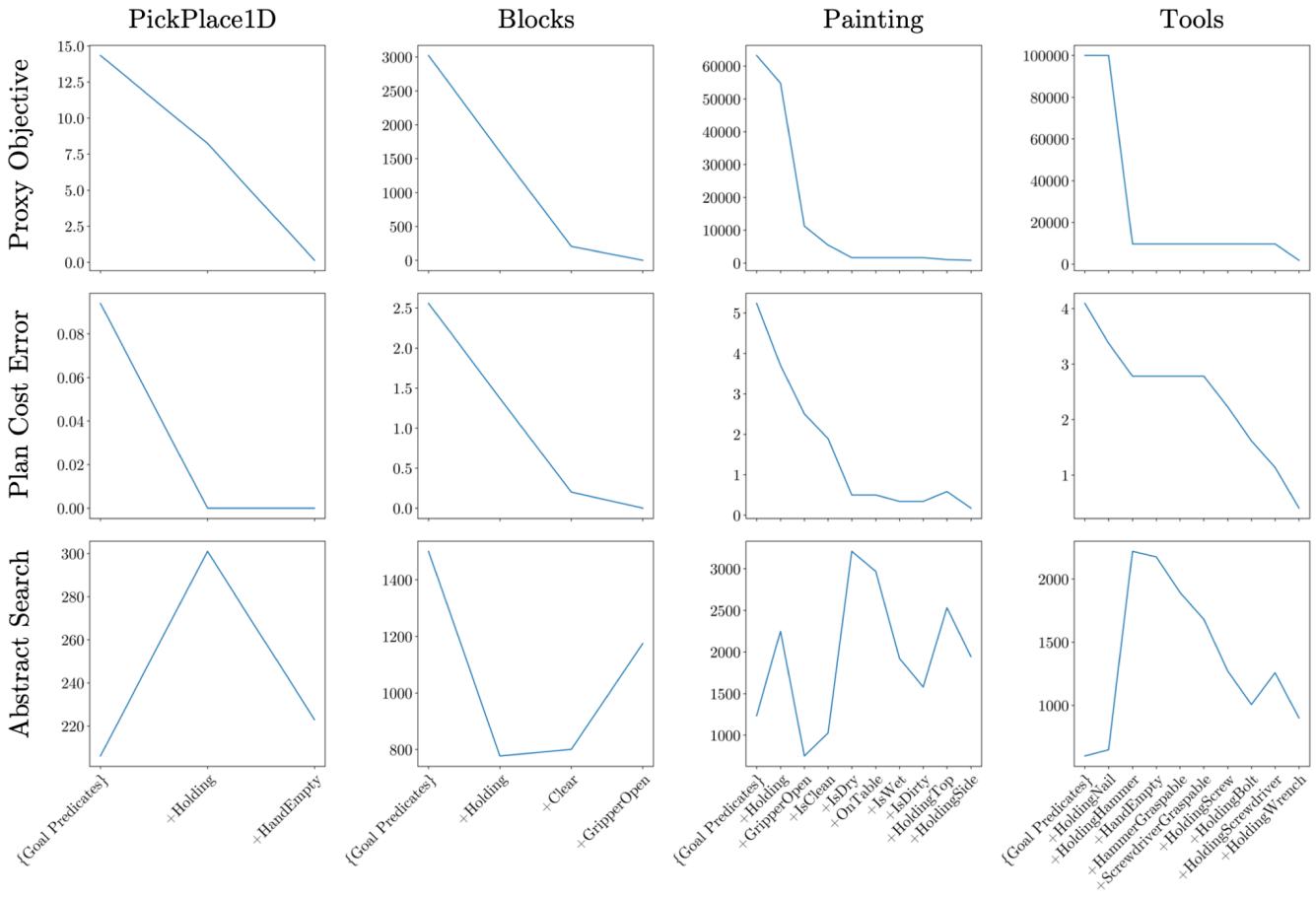


Figure 4: **Decomposing the surrogate objective.** In these plots, each column corresponds to one environment. The x-axes correspond to sets of manually designed predicates. The predicate sets grow in size from left to right, starting with the goal predicates alone, adding one predicate at each tick mark, and concluding with the full set of manual predicates for the respective environment. The order that the predicates are added was determined by hill climbing with respect to the surrogate objective. The top row shows the surrogate objective itself; the middle row shows the plan cost error $|\textsc{Cost}(\hat{\pi}) - \textsc{Cost}(\pi^*)|$ minimized over the first 8 skeletons generated by abstract search; and the bottom row shows the total number of nodes created by the abstract search (our measure of abstract search time), cumulative over the 8 skeletons. *There are two key takeaways from this plot.* (1) The surrogate objective (first row) monotonically decreases in all environments; this smoothness makes local search over candidate predicate sets an attractive option. (2) Neither of the two components that make up the surrogate objective — plan cost error (second row) or abstract search time (third row) — has the same monotonically decreasing property on its own, suggesting that both parts are necessary for making our predicate invention pipeline work. All results are means over 10 seeds.

```
(define (domain blocks)
(:predicates
    (on ?v0 ?v1)
    (ontable ?v0)
    (clear ?v0)
    (nothingabove ?v0)
    (notonanyblock ?v0)
    (handempty)
    (holding ?v0)
)
(:action pick-up
    :parameters (?x)
    :precondition (and
        (clear ?x)
        (nothingabove ?x)
        (notonanyblock ?x)
        (ontable ?x)
        (handempty))
    :effect (and
        (not (clear ?x))
        (not (ontable ?x))
        (not (handempty))
        (holding ?x))
)
(:action put-down
    :parameters (?x)
    :precondition (and
        (holding ?x)
        (nothingabove ?x)
        (notonanyblock ?x))
    :effect (and
        (clear ?x)
        (not (holding ?x))
        (handempty)
        (ontable ?x))
)
(:action stack
    :parameters (?x ?y)
    :precondition (and
        (holding ?x)
        (clear ?y)
        (nothingabove ?x)
        (notonanyblock ?x)
        (nothingabove ?y))
    :effect (and
        (not (holding ?x))
        (not (clear ?y))
        (clear ?x)
        (not (nothingabove ?y))
        (not (notonanyblock ?x))
        (handempty)
        (on ?x ?y))
)
(:action unstack
    :parameters (?x ?y)
    :precondition (and
        (on ?x ?y)
        (clear ?x)
        (nothingabove ?x)
        (handempty))
    :effect (and
        (holding ?x)
        (clear ?y)
        (not (clear ?x))
        (nothingabove ?y)
        (notonanyblock ?x)
        (not (handempty))
        (not (on ?x ?y)))
))
```

```
P1() ≜ (∀ ?x:block . (?x.grasp <= -0.485))
P2(?y:target) ≜ (∀ ?x:block . ¬Covers(?x, ?y))
P3(?x:block) ≜ (∀ ?y:target . ¬Covers(?x, ?y))
P4(?x:block) ≜ ¬(?x.grasp <= -0.485)
P5() ≜ ¬(∀ ?x:block . (?x.grasp <= -0.485))
```

```
Op0:
Parameters: [?x0:block, ?x1:target]
Preconditions:
 P3(?x0:block)
 P4(?x0:block)
 P5()
 P2(?x1:target)
Add Effects:
 Covers(?x0:block, ?x1:target)
 P1()
Delete Effects:
 P3(?x0:block)
 P4(?x0:block)
 P5()
 P2(?x1:target)
Controller: (PickPlace, [])
```

```
Op1:
Parameters: [?x0:block]
Preconditions:
 P3(?x0:block)
 P1()
Add Effects:
 P4(?x0:block)
 P5()
Delete Effects:
 P1()
Controller: (PickPlace, [])
```

Figure 5: PickPlace1D learned abstractions (top: predicates, bottom: operators).

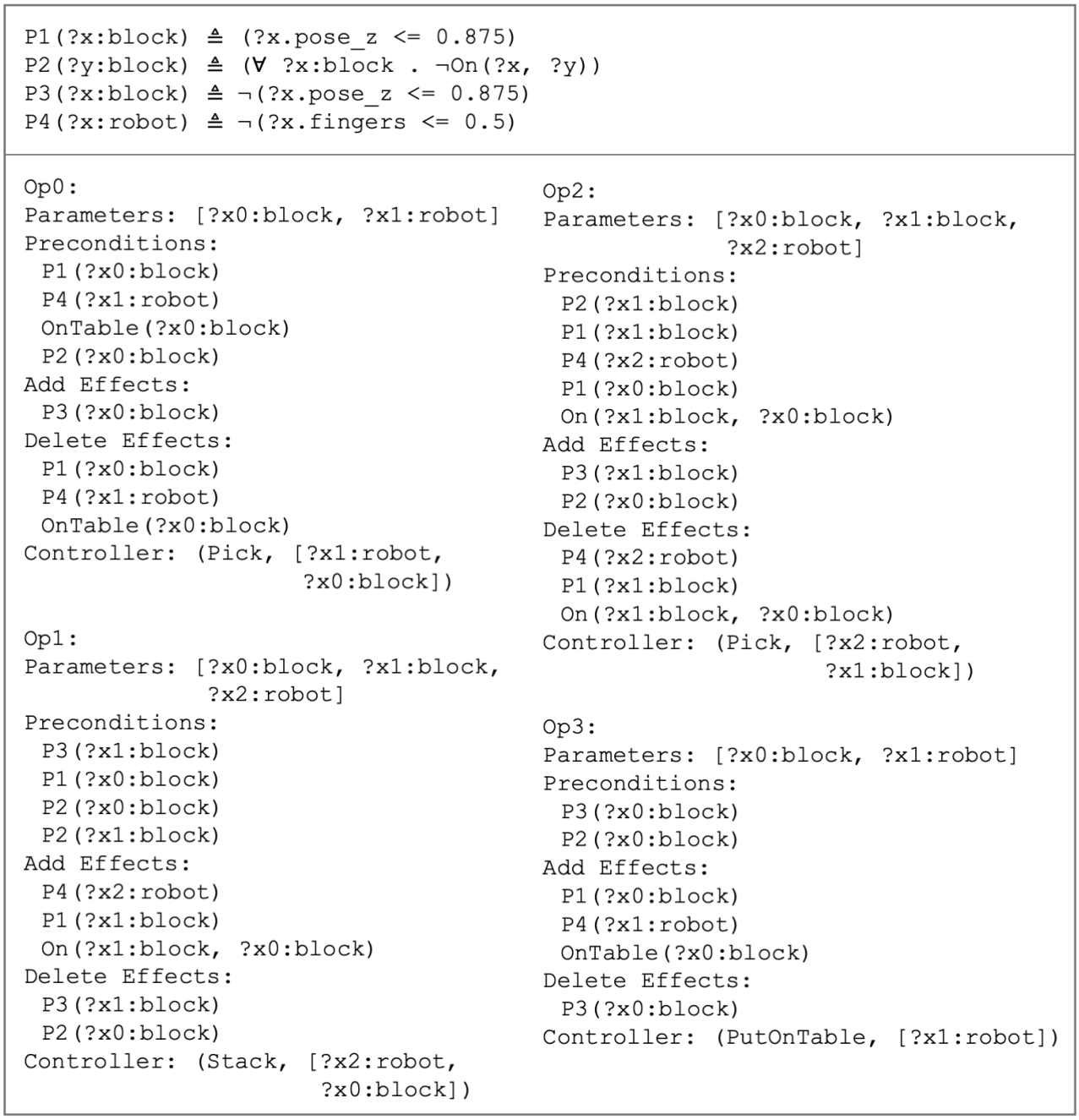


Figure 6: Blocks learned abstractions (top: predicates, bottom: operators).

```
P1(?x:obj) ≜ (?x.color <= 0.125)
P2(?x:obj) ≜ (?x.dirtiness <= 0.498)
P3(?x:obj) ≜ (?x.grasp <= 0.25)
P4(?x:obj) ≜ (?x.pose_y <= -0.306)
P5(?x:obj) ≜ (?x.wetness <= 0.5)
P6() ≜ (∀ ?x:lid . (?x.is_open <= 0.5))
P7() ≜ (∀ ?x:obj, ?y:box . ¬InBox(?x, ?y))
P8() ≜ (∀ ?x:obj, ?y:box . ¬IsBoxColor(?x, ?y))
P9() ≜ (∀ ?x:obj . ¬(?x.grasp <= 0.25))
P10(?x:obj) ≜ (∀ ?y:shelf . IsShelfColor(?x, ?y))
P11(?x:obj) ≜ ¬(?x.color <= 0.125)
P12(?x:obj) ≜ ¬(?x.dirtiness <= 0.498)
P13(?x:obj) ≜ ¬(?x.grasp <= 0.5)
P14(?x:obj) ≜ ¬(?x.grasp <= 0.25)
P15(?x:obj) ≜ ¬(?x.held <= 0.5)
P16(?x:obj) ≜ ¬(?x.wetness <= 0.5)
P17(?x:robot) ≜ ¬(?x.fingers <= 0.5)
P18() ≜ ¬(∀ ?x:lid . (?x.is_open <= 0.5))
P19(?x:obj) ≜ ¬(∀ ?y:box . IsBoxColor(?x, ?y))
P20(?x:obj, ?y:box) ≜ ¬InBox(?x, ?y)
```

```
Op0:
Parameters: [?x0:obj, ?x1:robot]
Preconditions:
  P19(?x0:obj)
  P4(?x0:obj)
  P12(?x0:obj)
  P15(?x0:obj)
  P5(?x0:obj)
  P1(?x0:obj)
Add Effects:
  P16(?x0:obj)
  P2(?x0:obj)
Delete Effects:
  P12(?x0:obj)
  P5(?x0:obj)
Controller: (Wash, [?x1:robot])

Op1:
Parameters: [?x0:obj, ?x1:robot]
Preconditions:
  P16(?x0:obj)
  P19(?x0:obj)
  P2(?x0:obj)
  P4(?x0:obj)
  P15(?x0:obj)
  P1(?x0:obj)
Add Effects:
  P5(?x0:obj)
Delete Effects:
  P16(?x0:obj)
Controller: (Dry, [?x1:robot])

Op2:
Parameters: [?x0:obj, ?x1:shelf,
             ?x2:robot]
Preconditions:
  P19(?x0:obj)
  P2(?x0:obj)
  P4(?x0:obj)
  P15(?x0:obj)
  P5(?x0:obj)
  P1(?x0:obj)
Add Effects:
  P10(?x0:obj)
  IsShelfColor(?x0:obj, ?x1:shelf)
  P11(?x0:obj)
Delete Effects:
  P1(?x0:obj)
Controller: (Paint, [?x2:robot])

Op3:
Parameters: [?x0:lid, ?x1:robot]
Preconditions:
  P6()
  P17(?x1:robot)
  P9()
  P7()
Add Effects:
  P18()
Delete Effects:
  P6()
Controller: (OpenLid, [?x1:robot,
                       ?x0:lid])
```

Figure 7: Painting learned abstractions (top: predicates, bottom: operators part 1 of 2).

```
Op4:
Parameters: [?x0:obj, ?x1:shelf, ?x2:robot]
Preconditions:
  P3(?x0:obj)
  P19(?x0:obj)
  P2(?x0:obj)
  IsShelfColor(?x0:obj, ?x1:shelf)
  P11(?x0:obj)
  P4(?x0:obj)
  P15(?x0:obj)
  P10(?x0:obj)
  P5(?x0:obj)
Add Effects:
  InShelf(?x0:obj, ?x1:shelf)
  P17(?x2:robot)
  P9()
  P14(?x0:obj)
Delete Effects:
  P4(?x0:obj)
  P15(?x0:obj)
  P3(?x0:obj)
Controller: (Place, [?x2:robot])

Op5:
Parameters: [?x0:obj, ?x1:robot]
Preconditions:
  P14(?x0:obj)
  P18()
  P9()
  P7()
  P4(?x0:obj)
  P17(?x1:robot)
Add Effects:
  P15(?x0:obj)
  P13(?x0:obj)
Delete Effects:
  P17(?x1:robot)
Controller: (Pick, [?x1:robot, ?x0:obj])

Op6:
Parameters: [?x0:obj, ?x1:box, ?x2:robot]
Preconditions:
  P19(?x0:obj)
  P2(?x0:obj)
  P7()
  P4(?x0:obj)
  P8()
  P20(?x0:obj, ?x1:box)
  P15(?x0:obj)
  P5(?x0:obj)
  P1(?x0:obj)
Add Effects:
  P11(?x0:obj)
  IsBoxColor(?x0:obj, ?x1:box)
Delete Effects:
  P8()
  P19(?x0:obj)
  P1(?x0:obj)
Controller: (Paint, [?x2:robot])

Op7:
Parameters: [?x0:obj, ?x1:robot]
Preconditions:
  P14(?x0:obj)
  P9()
  P7()
  P13(?x0:obj)
  P4(?x0:obj)
  P15(?x0:obj)
Add Effects:
  P17(?x1:robot)
Delete Effects:
  P15(?x0:obj)
  P13(?x0:obj)
Controller: (Place, [?x1:robot])

Op8:
Parameters: [?x0:obj, ?x1:box, ?x2:robot]
Preconditions:
  P2(?x0:obj)
  P14(?x0:obj)
  P18()
  P11(?x0:obj)
  P9()
  P7()
  P13(?x0:obj)
  P4(?x0:obj)
  P20(?x0:obj, ?x1:box)
  IsBoxColor(?x0:obj, ?x1:box)
  P15(?x0:obj)
  P5(?x0:obj)
Add Effects:
  InBox(?x0:obj, ?x1:box)
  P17(?x2:robot)
Delete Effects:
  P7()
  P13(?x0:obj)
  P4(?x0:obj)
  P20(?x0:obj, ?x1:box)
  P15(?x0:obj)
Controller: (Place, [?x2:robot])

Op9:
Parameters: [?x0:obj, ?x1:robot]
Preconditions:
  P19(?x0:obj)
  P14(?x0:obj)
  P9()
  P4(?x0:obj)
  P17(?x1:robot)
Add Effects:
  P15(?x0:obj)
  P3(?x0:obj)
Delete Effects:
  P17(?x1:robot)
  P9()
  P14(?x0:obj)
Controller: (Pick, [?x1:robot, ?x0:obj])

Op10:
Parameters: [?x0:obj, ?x1:robot]
Preconditions:
  P4(?x0:obj)
  P15(?x0:obj)
  P3(?x0:obj)
  P7()
Add Effects:
  P17(?x1:robot)
  P9()
  P14(?x0:obj)
Delete Effects:
  P15(?x0:obj)
  P3(?x0:obj)
Controller: (Place, [?x1:robot])
```

Figure 8: Painting learned abstractions (operators part 2 of 2).

```
P1(?x:robot) ≜ (?x.fingers <= 0.5)
P2(?x:screwdriver) ≜ (?x.size <= 0.503)
P3() ≜ (∀ ?x:nail, ?y:contraption . NailPlaced(?x, ?y))
P4() ≜ (∀ ?x:screw, ?y:contraption . ¬ScrewPlaced(?x, ?y))
P5() ≜ (∀ ?x:screw . ¬ScrewFastened(?x))
P6(?x:bolt) ≜ ¬(?x.is_held <= 0.5)
P7(?x:hammer) ≜ ¬(?x.is_held <= 0.5)
P8(?x:nail) ≜ ¬(?x.is_held <= 0.5)
P9(?x:robot) ≜ ¬(?x.fingers <= 0.5)
P10(?x:screw) ≜ ¬(?x.is_held <= 0.5)
P11(?x:screwdriver) ≜ ¬(?x.is_held <= 0.5)
P12(?x:wrench) ≜ ¬(?x.is_held <= 0.5)
P13() ≜ ¬(∀ ?x:bolt . BoltFastened(?x))
P14() ≜ ¬(∀ ?x:bolt . ¬BoltFastened(?x))
P15() ≜ ¬(∀ ?x:nail, ?y:contraption . NailPlaced(?x, ?y))
P16() ≜ ¬(∀ ?x:screw, ?y:contraption . ¬ScrewPlaced(?x, ?y))
P17() ≜ ¬(∀ ?x:screw . ScrewFastened(?x))
P18(?x:bolt) ≜ ¬(∀ ?y:contraption . ¬BoltPlaced(?x, ?y))
```

```
Op0:
Parameters: [?x0:nail, ?x1:robot]
Preconditions:
 P9(?x1:robot)
 P15()
Add Effects:
 P12(?x0:nail)
 P1(?x1:robot)
Delete Effects:
 P9(?x1:robot)
Controller: (PickNail, [?x1:robot,
                        ?x0:nail])

Op1:
Parameters: [?x0:contraption,
             ?x1:nail, ?x2:robot]
Preconditions:
 P1(?x2:robot)
 P12(?x1:nail)
 P15()
Add Effects:
 P9(?x2:robot)
 NailPlaced(?x1:nail,
            ?x0:contraption)
Delete Effects:
 P1(?x2:robot)
 P12(?x1:nail)
Controller: (Place, [?x2:robot])

Op2:
Parameters: [?x0:hammer, ?x1:robot]
Preconditions:
 P9(?x1:robot)
 P15()
Add Effects:
 P1(?x1:robot)
 P12(?x0:hammer)
Delete Effects:
 P9(?x1:robot)
Controller: (PickHammer, [?x1:robot,
                          ?x0:hammer])

Op3:
Parameters: [?x0:contraption,
             ?x1:hammer, ?x2:nail,
             ?x3:robot]
Preconditions:
 NailPlaced(?x2:nail,
            ?x0:contraption)
 P12(?x1:hammer)
 P15()
 P1(?x3:robot)
Add Effects:
 NailFastened(?x2:nail)
Delete Effects:
Controller: (FastenNailWithHammer,
  [?x3:robot, ?x2:nail, ?x1:hammer,
   ?x0:contraption])
```

Figure 9: Tools learned abstractions (top: predicates, bottom: operators part 1 of 2).

```
Op4:
Parameters: [?x0:bolt, ?x1:robot]
Preconditions:
   P13()
   P9(?x1:robot)
Add Effects:
   P1(?x1:robot)
   P12(?x0:bolt)
Delete Effects:
   P9(?x1:robot)
Controller: (PickBolt, [?x1:robot, ?x0:bolt])

Op5:
Parameters: [?x0:bolt, ?x1:contraption, ?x2:robot]
Preconditions:
   P13()
   P1(?x2:robot)
   P12(?x0:bolt)
Add Effects:
   BoltPlaced(?x0:bolt, ?x1:contraption)
   P18(?x0:bolt)
   P9(?x2:robot)
Delete Effects:
   P1(?x2:robot)
   P12(?x0:bolt)
Controller: (Place, [?x2:robot])

Op6:
Parameters: [?x0:hammer, ?x1:robot]
Preconditions:
   P1(?x1:robot)
   P15()
   P5()
   P12(?x0:hammer)
Add Effects:
   P9(?x1:robot)
Delete Effects:
   P1(?x1:robot)
   P12(?x0:hammer)
Controller: (Place, [?x1:robot])

Op7:
Parameters: [?x0:robot, ?x1:wrench]
Preconditions:
   P13()
   P9(?x0:robot)
Add Effects:
   P12(?x1:wrench)
   P1(?x0:robot)
Delete Effects:
   P9(?x0:robot)
Controller: (PickWrench, [?x0:robot, ?x1:wrench])

Op8:
Parameters: [?x0:bolt, ?x1:contraption, ?x2:robot, ?x3:wrench]
Preconditions:
   P18(?x0:bolt)
   P13()
   P12(?x3:wrench)
   BoltPlaced(?x0:bolt, ?x1:contraption)
   P1(?x2:robot)
Add Effects:
   BoltFastened(?x0:bolt)
   P14()
Delete Effects:
   P13()
Controller: (FastenBoltWithWrench, [?x2:robot, ?x0:bolt, ?x3:wrench,
                                    ?x1:contraption])

Op9:
Parameters: [?x0:robot, ?x1:screw]
Preconditions:
   P4()
   P5()
   P9(?x0:robot)
   P17()
Add Effects:
   P12(?x1:screw)
   P1(?x0:robot)
Delete Effects:
   P9(?x0:robot)
Controller: (PickScrew, [?x0:robot, ?x1:screw])

Op10:
Parameters: [?x0:contraption, ?x1:robot, ?x2:screw]
Preconditions:
   P4()
   P1(?x1:robot)
   P12(?x2:screw)
   P5()
   P17()
Add Effects:
   P9(?x1:robot)
   ScrewPlaced(?x2:screw, ?x0:contraption)
   P16()
Delete Effects:
   P4()
   P1(?x1:robot)
   P12(?x2:screw)
Controller: (Place, [?x1:robot])

Op11:
Parameters: [?x0:robot, ?x1:wrench]
Preconditions:
   P1(?x0:robot)
   P12(?x1:wrench)
   P5()
   P14()
Add Effects:
   P9(?x0:robot)
Delete Effects:
   P12(?x1:wrench)
   P1(?x0:robot)
Controller: (Place, [?x0:robot])

Op12:
Parameters: [?x0:robot, ?x1:screwdriver]
Preconditions:
   P16()
   P17()
   P5()
   P2(?x1:screwdriver)
   P9(?x0:robot)
Add Effects:
   P1(?x0:robot)
   P12(?x1:screwdriver)
Delete Effects:
   P9(?x0:robot)
Controller: (PickScrewdriver, [?x0:robot, ?x1:screwdriver])

Op13:
Parameters: [?x0:contraption, ?x1:robot, ?x2:screw, ?x3:screwdriver]
Preconditions:
   P12(?x3:screwdriver)
   P16()
   P1(?x1:robot)
   P2(?x3:screwdriver)
   ScrewPlaced(?x2:screw, ?x0:contraption)
   P5()
   P17()
Add Effects:
   ScrewFastened(?x2:screw)
Delete Effects:
   P5()
   P17()
Controller: (FastenScrewWithScrewdriver, [?x1:robot, ?x2:screw,
                                          ?x3:screwdriver, ?x0:contraption])

Op14:
Parameters: [?x0:contraption, ?x1:robot, ?x2:screw]
Preconditions:
   P16()
   ScrewPlaced(?x2:screw, ?x0:contraption)
   P17()
   P5()
   P9(?x1:robot)
Add Effects:
   ScrewFastened(?x2:screw)
Delete Effects:
   P5()
   P17()
Controller: (FastenScrewByHand, [?x1:robot, ?x2:screw, ?x0:contraption])

Op15:
Parameters: [?x0:bolt, ?x1:contraption, ?x2:robot, ?x3:wrench]
Preconditions:
   P18(?x0:bolt)
   P4()
   P13()
   P3()
   P5()
   P12(?x3:wrench)
   BoltPlaced(?x0:bolt, ?x1:contraption)
   P1(?x2:robot)
Add Effects:
   BoltFastened(?x0:bolt)
   P14()
Delete Effects:
Controller: (FastenBoltWithWrench, [?x2:robot, ?x0:bolt, ?x3:wrench,
                                    ?x1:contraption])

Op16:
Parameters: [?x0:bolt, ?x1:contraption, ?x2:robot, ?x3:wrench]
Preconditions:
   P18(?x0:bolt)
   P14()
   P4()
   P13()
   P3()
   P5()
   P12(?x3:wrench)
   BoltPlaced(?x0:bolt, ?x1:contraption)
   P1(?x2:robot)
Add Effects:
   BoltFastened(?x0:bolt)
Delete Effects:
   P13()
Controller: (FastenBoltWithWrench, [?x2:robot, ?x0:bolt, ?x3:wrench,
                                    ?x1:contraption])

Op17:
Parameters: [?x0:robot, ?x1:screwdriver]
Preconditions:
   P2(?x1:screwdriver)
   P12(?x1:screwdriver)
   P1(?x0:robot)
   P16()
Add Effects:
   P9(?x0:robot)
Delete Effects:
   P1(?x0:robot)
   P12(?x1:screwdriver)
Controller: (Place, [?x0:robot])
```

Figure 10: Tools learned abstractions (operators part 2 of 2).