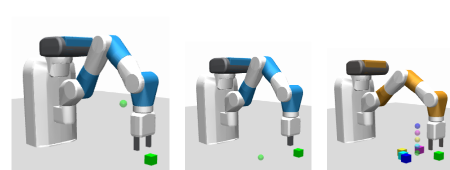
Abstraction Refinement-guided Program Synthesis for Robot Learning from Demonstrations

GUOFENG CUI, Rutgers University, USA
YUNING WANG, Rutgers University, USA
WENSEN MAO, Rutgers University, USA
YUANLIN DUAN, Rutgers University, USA
HE ZHU, Rutgers University, USA

Over the past decade, deep reinforcement learning (RL) techniques have significantly advanced robotic systems. However, due to the complex architectures of neural network models, ensuring their trustworthiness is a considerable challenge. *Programmatic reinforcement learning* has surfaced as a promising approach to improve interpretability by using *domain-specific programs* to represent RL models. Nonetheless, synthesizing robot-control programs remains challenging. Existing methods rely on domain-specific languages (DSLs) populated with user-defined state abstraction predicates and a library of low-level controllers (e.g., raising a robot's end effector up) as abstract actions to boot synthesis, which is impractical in unknown environments that lack such predefined components. To address this limitation, we introduce RoboScribe, a novel abstraction refinement guided program synthesis framework that automatically derives robot state and action abstractions from raw, unsegmented task demonstrations in high-dimensional, continuous spaces. It iteratively enriches and refines an initially coarse abstraction until it generates a task-solving program over the abstracted robot environment. RoboScribe is effective in synthesizing iterative programs by inferring recurring subroutines directly from the robot's raw, continuous state and action spaces, without needing predefined abstractions. Experimental results show that RoboScribe programs inductively generalize to long-horizon robot tasks involving arbitrary numbers of objects, outperforming baseline methods in terms of both interpretability and efficiency.

1 Introduction



(a) Pick&place (b) Push (c) Tower

Fig. 1. Robot environments for various tasks.

Learning-enabled systems, which incorporate machine learning components to learn from data or sensor inputs, are increasingly used in large-scale applications. In the robotics domain, deep reinforcement learning (RL) techniques have shown promise in developing intelligent agents for robot control, offering robust alternatives to analytical models in adaptive control systems. These methods enable an agent to develop complex skills through environmental interaction. However, ensuring the trustworthiness of deep RL systems is challenging due to the intricate nature of neural network structures, the high dimensionality of data they process, and the unpredictable variations in real-world environments. To address this, *programmatic reinforcement learning* has emerged, focusing on synthesizing *domain-specific programs* as RL model representations to enhance interpretability [5, 24, 26, 45, 51, 58, 60, 61, 63, 64]. For example, PROLEX [42] and Tabula [44] learn robot-control programs from task demonstrations, generalizing a specific sequence of high-level control actions to a general program to solve unseen tasks. A recent work, ReGuS (reward-guided synthesis) [12], generates robot-control programs directly from reward signals and demonstrates that synthesizing programs with rich control-flow constructs (e.g., loops) can effectively tackle long-horizon and sparse-reward tasks that often confound deep RL techniques.

Authors' Contact Information: Guofeng Cui, gc669@cs.rutgers.edu, Rutgers University, USA; Yuning Wang, yw895@rutgers.edu, Rutgers University, USA; Wensen Mao, wm300@rutgers.edu, Rutgers University, USA; Yuanlin Duan, yuanlin.duan@rutgers.edu, Rutgers University, USA; He Zhu, hz375@cs.rutgers.edu, Rutgers University, USA.

State and Action Abstraction. Prior work that scales program synthesis techniques to high-dimensional robotics environments with continuous state and action spaces often crafts a domain-specific language (DSL) integrating *state abstraction predicates* and *abstract control actions*. To illustrate the need for abstraction, consider the `Pick&Place` robot task shown in Fig. 1a, where a manipulator must pick up a block (green) from a table and place it in a target position (sphere) in mid-air. The robot used is a 7-DoF Fetch Mobile Manipulator with a two-fingered parallel gripper. The robot's state space includes kinematic information of the block and the end effector including the Cartesian coordinates of the desired final block position. Its action space represents the Cartesian displacement of the end effector to set to and the positional displacement of each finger of the gripper. Directly generating programs in a high-dimensional continuous space poses significant challenges for a synthesizer due to the complexity and nature of the space involved.

State abstraction predicates [2, 17] construct a higher-level representation of the robot's environment based on observed sensor data. This higher-level representation can then be reasoned about using standard language constructs, such as loops and conditionals, to trigger suitable actions from a current state. An abstract action c is an application of a low-level skill of the robot. These skills can be derived from either robot APIs or pre-trained neural network controllers. Conceptually, skills are modular and reusable, and can be likened to building blocks or subroutines that contribute to the overall control policy of the robot in diverse contexts. For example, for `Pick&Place`, ReGuS [12] involves state abstraction predicates `Near` (indicates if the gripper is close to the block b), `Holding` (indicates if the gripper is holding b), `Above` (indicates if the gripper is above b), and `At` (indicates if b is in the goal region $g \in \mathbb{R}^3$) into the DSL used to search a program to address this task. The DSL also includes a set of abstract actions to operate the robot in the continues environment for `openGripp()`, `closeGripp()`, `moveUp()`, `moveDown()`, `move(`g`)` that moves the gripper to a goal region g. ReGuS synthesizes a program, as shown in Fig. 2, that guides the robotic gripper to move above the block b in the environment, lower to grab it, and then transport the block to the designated goal area g.

```
while(not At(b, g)):
 if (Above(b)):
  if (Near(b)):
   if (Holding(b)):
    move(g)
   else:
    closeGripp()
  else:
   openGripp()
   if(not Near(b)):
    moveDown()
 else:
  move(b)
```

Fig. 2. Synthesized program for pick&place by ReGuS [12].

Challenges Faced by Existing Approaches. Several challenges remain in advancing robot-control program synthesis in real-world scenarios.

- (*i*) A crucial bottleneck in existing synthesis techniques is the reliance on manually designed state abstraction predicates and action abstraction in a DSL to bootstrap synthesis. High-quality abstractions ensure that programs synthesized from them can be executed successfully in the real environment but often require significant human effort and domain knowledge to customize effectively. For example, the state and action abstractions for the `Pick&place` task described above cannot be applied to a robot with a push-only gripper as visualized in the `Push` task in Fig. 1b. This type of gripper is designed to apply force to reposition objects on a surface, but it cannot grasp or lift them. Automatically learning state and action abstractions has been a key area of research in task and motion planning for robot control [6, 21, 27, 41]. Existing techniques often have significant limitations, as they either learn predicates from demonstrations while assuming that low-level controllers are already available [13, 34, 53], or learn low-level controllers from demonstrations while assuming that the necessary predicates are predefined [1, 15, 52]. Simultaneous discovery of both state abstraction predicates and abstract actions remains a significant challenge [32, 52, 53].

- (*ii*) Existing approaches encounter difficulties when synthesizing programs that generalize to tasks involving repeated subroutines or arbitrary numbers of objects, such as directing a robot arm to stack scattered objects into a pyramid. These tasks often require complex control-flow structures like state-conditioned loops. When a domain-specific language (DSL) with predefined state abstraction and low-level controllers is available, as in existing work, this complexity is reduced. For example, in PROLEX [42], task demonstrations are represented as sequences of calls to user-defined low-level controllers rather than continuous, raw actions in the robot's operational space. This representation enables it to deduce when and how these controllers are employed repeatedly or conditionally. Consequently, it can extract high-level control structures, such as loops and conditional statements, from observed tasks. Synthesizing loop programs for robot control in high-dimensional, continuous state and action spaces without predefined abstractions is still a major hurdle.

This Paper. To make program synthesis feasible for high-dimensional, continuous state and action spaces in robotic environments, our main idea is to develop ***abstraction refinement*** techniques that automatically generates appropriate state and action abstractions as part of the synthesis process. To this end, we develop **RoboScribe**, a novel *abstraction refinement-guided program synthesis* framework. RoboScribe is visualized in Fig. 3 and is based on the following two key ideas:

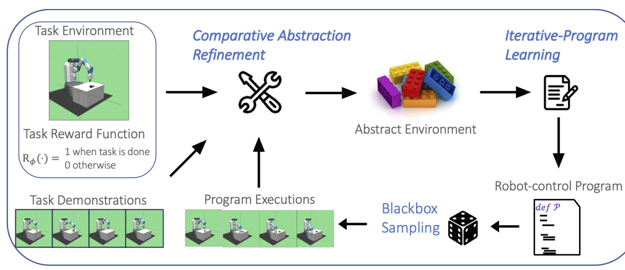


Fig. 3. Overview of the RoboScribe framework.

(1) Comparative Abstraction Refinement: To overcome challenge (*i*), RoboScribe iteratively refines an initially coarse abstraction of the robot environment until a valid program capable of solving the task is synthesized from the abstract environment. The initial coarse abstraction differentiates only between states that have met the (unknown) goal condition ψ_R in the environment and those that have not:

$$\texttt{True} \rightsquigarrow \psi_R$$

under the assumption that a single low-level controller as the target program can be learned to transition all possible initial states (`True`) to the goal condition (ψ_R). In the context of `pick&place`, the initial abstraction only identifies goal states where the block is successfully placed at the target position. If this assumption fails, RoboScribe incrementally learns state abstraction predicates that capture critical intermediate states toward task completion. For example, in the `pick&place` task, the robot must move its gripper close to the block, grasp it, and lift it to a designated position. By learning state abstraction predicates that identify such key subgoal states and abstract actions as low-level controllers that transition the agent across these subgoals, RoboScribe accurately captures the task's hierarchical structure. We posit that state abstraction predicates can be learned if demonstrations showing task completion are provided to the synthesizer. Systematically comparing states within these demonstrations to those observed during failed robot behaviors which are learned in the real environment allows RoboScribe to discern necessary intermediate states in the demonstrations pivotal for the task's success. For `pick&place`, a low-level controller trained to fulfill ψ_R may struggle with grasping the block first, as exemplified in the program execution behavior shown in Fig. 3. Leveraging insights gained from successful demonstrations, RoboScribe refines the abstraction with a predicate φ that captures the states where the robot's gripper is positioned close to the block:

$$\texttt{True} \rightsquigarrow \varphi \rightsquigarrow \psi_R$$

The refined abstraction adds a crucial step that bridges the initial and goal states, breaking down the complex task into manageable subtasks. This process is recursively conducted, enabling the task to be eventually solved through a series of progressively refined subtasks that lead to ψ_R.

(2) Iterative-Program Learning: To address challenge (*ii*), RoboScribe leverages state abstraction predicates learned *on-the-fly* to identify repeating subroutines within demonstrations. For instance, consider the `Tower` task of manipulating a robot arm to stack blocks into a tower visualized in Fig. 1c. Key state abstraction predicates for subgoals like grasping a block and lifting it to a specific position, learned for handling one block, recur throughout the demonstration for handling other blocks. RoboScribe exploits repeated subgoal predicates in demonstrations to construct the *loop body* that guides the agent through transitions between the subgoals vian abstract actions in the form of low-level controllers. However, the order in which recurring objects are addressed across loop iterations remains unspecified. This ordering is crucial in tasks like `Tower`, where placing a block in the goal position before positioning the underlying blocks leads to failed outcomes. RoboScribe synthesizes a predicate that identifies the correct object order by analyzing the rationale behind the demonstrations, explaining why certain objects are handled before others. In a nutshell, to synthesizing programs that generalize to tasks involving arbitrary numbers of objects, RoboScribe first constructs a loop program skeleton, then fills in an object ordering predicate as the loop guard to determine the sequence for handling objects, and finally optimizes action abstractions to ensure robust generalization across all loop iterations.

Evaluation. We implemented RoboScribe and evaluated it using a benchmark suite of complex robot object manipulation tasks. Experimental results demonstrate that RoboScribe programs can inductively generalize to long-horizon tasks involving arbitrary numbers of objects, outperforming baseline methods in both interpretability and efficiency. For example, the programmatic agent synthesized by RoboScribe can efficiently use a robot arm to stack multiple blocks on a cluttered tabletop into a tower, a task known for its complexity in RL due to the need to handle long-horizon planning and precise manipulation [30, 39]. The agent can generalize to unseen configurations, such as placing blocks into multiple towers with zero-shot success.

Contributions. To summarize, this paper makes the following key contributions:

- We propose RoboScribe, a novel abstraction refinement technique that automatically derives robot state and action abstractions from raw, unsegmented task demonstrations to enable robot-control program synthesis in high-dimensional, continuous spaces.
- We develop an effective loop program synthesis algorithm that scales RoboScribe to long-horizon tasks involving unbounded environment objects. The algorithm excels in inferring repeating subroutines directly from demonstrations in the robot's raw, continuous state and action spaces.
- We evaluate RoboScribe in complex robot manipulation tasks, highlighting its effectiveness in learning and generalizing control strategies.

2 Overview

In this section, we motivate the problem and provide an overview of our approach.

2.1 Key Assumptions

Object-centric Views. Following common practice in robotics [25, 31, 50, 55, 56, 66], RoboScribe assumes that the robot receives an object-centric view $\{\boldsymbol{\mu}, \boldsymbol{e}_1, \ldots, \boldsymbol{e}_N\}$ at each timestep (defined in Sec. 3). This view segments the world into discrete objects and classifies them into categories based on sensor data, where $\boldsymbol{\mu}$ represents the robot's end effector and $\boldsymbol{e}_i$ denotes an entity in the

$$
\begin{array}{rl}
\text{Type } \tau ::= \{\text{EE, block, handle, mug, faucet}, \ldots\} & \text{Variable } V ::= \mu, v \\
\text{Expression } \alpha ::= & \mu \mid v \mid g(v) \mid \alpha \downarrow_{x,y,z} \mid \alpha \downarrow_{x,y} \mid \alpha \downarrow_{x} \mid \alpha \downarrow_{y} \mid \alpha \downarrow_{z} \mid \alpha - \alpha \mid \|\alpha\| \mid \mathtt{arctan2}(\alpha) \\
\text{Predicate } P ::= & \alpha < \alpha \mid \alpha < \phi \mid \alpha > \phi \mid \alpha \neq \mathit{null} \mid \text{true} \mid \neg P \mid P \wedge P \mid P \vee P \mid \exists v.\, P \\
\text{Action } c ::= & \pi_{\theta_1}(\mu, \{v\}, g(\{v'\})) \mid \pi_{\theta_2}(\mu, \{v\}, g(\{v'\})) \mid \ldots \qquad \pi_{\theta_i} \in \Pi_{\text{NN}} \\
\text{Statement } S ::= & \textbf{while } (P)\ \{S\} \mid \textbf{if } (P)\ S_1 \textbf{ else } S_2 \mid S_1; S_2 \mid v := \textbf{get}(\lambda v : \tau.\, P) \mid c \\
\text{Program } ::= & \textbf{def } \mathcal{P}\ (\mu : \text{EE}) : S
\end{array}
$$

Fig. 4. The Context-free grammar for the RoboScribe DSL $\mathcal{L}$, where EE refers to the robot's end effector.

environment. Each object $\boldsymbol{\mu}$ or $\boldsymbol{e_i}$ is associated with attributes such as its class and 3D position. Such views can be constructed using e.g. object detection [35, 36, 48] or discovery [16, 33] methods.

Goal-Directed Robot Tasks. We consider goal-directed robotic tasks where a goal function $\boldsymbol{g}$ maps entities e_i to their target regions. For each control task, $\boldsymbol{g}$ may be randomly generated. For example, in the `Pick&Place` task (Fig. 1a), the goal region (green sphere) can be randomly placed within the robot's workspace. The robot's objective is defined by a predicate $\boldsymbol{\varphi_R}$, which is true when a designated subset of entities has reached their final target poses and false otherwise—e.g., an entity is considered to have reached its goal if its distance to the target is below a threshold. In RoboScribe, the agent does not require the analytical form of $\boldsymbol{\varphi_R}$ but can query it to verify whether a state s satisfies $\boldsymbol{\psi_R}$.

Demonstrations. We assume that we can utilize supervision of a limited amount of task demonstrations for robot-control program synthesis. RoboScribe only assumes unsegmented task demonstrations as sequences of states $(s_0, s_1, \ldots, s_H)$ where s_0 is an initial state and each state s_i at timestep i presents an object-centric view of the system (i.e., a collection of objects in the scene and their attributes). For each demonstration, we assume that $\psi_R(s_H)$ is true.

2.2 Program and Domain-Specific Language

RoboScribe synthesizes robot-control programs using a generic DSL $\mathcal{L}$ in Fig. 4. In a program, the variable μ binds to the robot's end effector and a variable v binds to an object e_i in the robot's object-centric view, with its type τ determined by the object it references. Each object has attributes, such as its 3D pose estimated from sensors, denoted $v \downarrow_{x,y,z}$, where $\downarrow$ extracts attribute values from the object referenced by v. In Fig. 4, we enumerate attributes related to position in $\mathbb{R}^3$, but the DSL can be extended to include full attributes, including orientation, such as quaternions. As discussed in Sec. 2.1, leveraging the goal function g, $g(v)$ represents the goal region of the object referenced by v, and $\|\cdot\|$ represents the Euclidean norm.

State abstraction predicates P in $\mathcal{L}$ enable the robot to locate relevant objects, constructing a higher-level representation of its environment. The parameter ϕ in our predicates is a constant (vector) that must be learned. Our DSL supports existential quantifiers $\exists v$ in predicates P to identify objects meeting specific criteria, such as blocks below a certain height. Predicates can be used to define spatial relationships (e.g., using $\|\cdot\|$) and physical orientations (e.g., using `arctan2`) among environment objects, including the robot itself. For instance, `arctan2` can be used to encode alignment constraints between the end effector, a block, and its goal region for direct pushing.

Abstract actions c represent low-level controllers that encode the robot's capabilities in continuous environments. Each abstract action $c \in \Pi_{\text{NN}}$ is a deep neural network policy π_θ with trainable parameters θ. The policy π_θ is goal-conditioned. It takes as input the attribute values of the end effector μ and some objects referenced by $\{v\}$, as well as the goal regions of some objects referenced by $\{v'\}$, producing a control action suitable for execution in the raw environment.

Our DSL $\mathcal{L}$ also includes *state-conditioned loops* and *conditional statements*. The *assignment* statement $v := \textbf{get}(\lambda v : \tau.\ P)$ binds variable v to an object e_i of type τ in the robot's object-centirc view such that $P[e_i/v]$ holds. A RoboScribe program $\mathcal{P}$ is a function that takes a binding to the robot's end effector as input and executes a defined statement as its body.

It is important to highlight that the RoboScribe DSL $\mathcal{L}$ does *not* include predefined low-level controllers. **The set of low-level controllers Π_{NN} starts out *empty*.** The synthesis process inherently involves *learning appropriate state abstraction predicates P and constructing abstract control actions* Π_{NN} as a fundamental component.

2.3 Demonstration-directed Robot Environment Abstraction Refinement

Key Insight. Our key idea is to systematically compare states within successful task demonstrations with those from failed robot behaviors learned in the real environment, aiming to identify key subgoal states that are essential for task success.

Comparative Abstraction Refinement. We define a robot task $\mathcal{T}$ as $\mathcal{T}$: $\texttt{True} \rightsquigarrow \psi_R$ with the expected behavior of directing the robot to transition from arbitrary initial states $\texttt{True}$ (underlying some unknown initial environment state distribution) to states that satisfy the goal condition ψ_R. The initial abstraction $\texttt{True} \rightsquigarrow \psi_R$ is coarse, distinguishing only between successful states that meet the goal condition ψ_R and those that have not yet reached it, and assumes that a single low-level controller π_{ψ_R} can fully solve the task. RoboScribe attempts to learn π_{ψ_R} using an off-the-shelf deep RL algorithm driven by a task reward function that assigns a reward of 1.0 to any state s where $\psi_R(s)$ is true and 0 otherwise. In this sparse reward setting, trajectories induced by π_{ψ_R} often fail to encounter any positive feedback, resulting in learning failure. RoboScribe compares the successful behavior given in a set of demonstrations $\mathcal{D}$ with that of the learned controller to identify key differences that are essential for enabling task success.

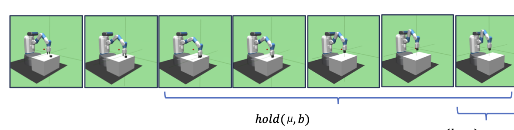


Fig. 5. Pick&place Demonstration.

For example, in the $\texttt{pick\&place}$ demonstration shown in Fig. 5, The end effector (referred to as the robot μ for simplicity) first holds the block and then places it on the target. However, the learned π_{ψ_R} struggles with grasping the block due to the absence of an explicit learning signal for this action. We extract states preceding successful task completion from demonstrations, forming a set $P_s = \{s_t \mid s_t \not\models \psi_R \wedge s_{t+1} \models \psi_R\}$, and compare them with states N_s collected from the learned controller's trajectories. RoboScribe synthesizes a state abstraction predicate to distinguish P_s and N_s to learn what prerequisite conditions are necessary for reaching ψ_R. We learn state abstraction predicates P derivable from the grammar in our DSL $\mathcal{L}$ in Fig. 4 based on Decision Tree (DT) learning. A DT is a binary tree that represents a Boolean formula. Each leaf of the tree is labeled either positive or negative for a subset of the samples in $P_s \cup N_s$. Each inner node is labeled by a decision of the form $\alpha \leq \phi$ where α is a feature and ϕ is a (learned) threshold. In our context, α is an expression derivable from the production rules for α in $\mathcal{L}$. We formalize the learning algorithm in Sec. 4.2.1. In this example, RoboScribe may learn a predicate $\texttt{hold}(\mu, b) \equiv \left\| \mu \downarrow_{x,y,z} - b \downarrow_{x,y,z} \right\| \leq \phi$ that defines states where a block b is being grasped by the robot gripper μ. Here, b references to the block in the environment. Using this learned predicate, we refine the initial abstraction as:

$$\texttt{True} \rightsquigarrow \texttt{hold}(\mu, b) \rightsquigarrow \psi_R \tag{1}$$

which effectively decomposes the task into subtasks: (1) $\mathcal{T}_1$: $\texttt{True} \rightsquigarrow \texttt{hold}(\mu, b)$ for reaching states where the block is grasped, and (2) $\mathcal{T}_2$: $\texttt{hold}(\mu, b) \rightsquigarrow \psi_R$ for achieving the goal condition after grasping the block. This process is recursively conducted until the task can be solved through a series of progressively refined subtasks to ψ_R. For example, for $\mathcal{T}_2$, RoboScribe may further learn a predicate $\texttt{at}(b, g(b))$ capturing states where the block is at the goal region, along with two subtasks: $\mathcal{T}_{2_1}$: $\texttt{hold}(\mu, b) \rightsquigarrow \texttt{at}(b, g(b))$ for moving the block towards its goal region, and $\mathcal{T}_{2_2}$: $\texttt{at}(b, g(b)) \rightsquigarrow \psi_R$ for maintaining the block in the target position of the task, resulting in the following refined abstraction:

$$\texttt{True} \rightsquigarrow \texttt{hold}(\mu, b) \rightsquigarrow \texttt{at}(b, g(b)) \rightsquigarrow \psi_R \tag{2}$$

Abstract Subtask Tree. RoboScribe structures environment abstractions as abstract subtask trees, formalized in Sec. 4. Each tree node φ or ψ encodes a state abstraction predicate that defines a subgoal condition. Each (inverted) tree edge $\varphi \rightsquigarrow \psi$ represents an abstract action, to be grounded as a low-level controller in the real environment, that fulfills the subtask of transitioning any state within φ to a subgoal state in ψ, guiding the agent toward the completion of its overall task goal condition ψ_R at the root of the tree. Tree representations effectively capture the structure of multi-goal tasks involving multiple objects, providing a clear framework for task decomposition and execution. To satisfy a subgoal ψ, the agent must complete all the subgoals of its predecessors φ such that $\varphi \rightsquigarrow \psi$. Each subgoal φ in the predecessors of ψ corresponds to the manipulation of a distinct object. For example, consider the PlaceCubeDrawer task depicted in Fig. 6 left. The goal of the Sawyer robot in this task is to pick up a cube b from the desk of a cabinet and place it inside the drawer below. The abstract subtask tree for this task is shown in Fig. 6 (right). The agent must complete the following subtasks: first moving the gripper μ near the cabinet door handle h, latching the handle, pulling it to align with the goal position $g(b)$ for the cube b, and then holding b before placing it inside the drawer at $g(b)$.

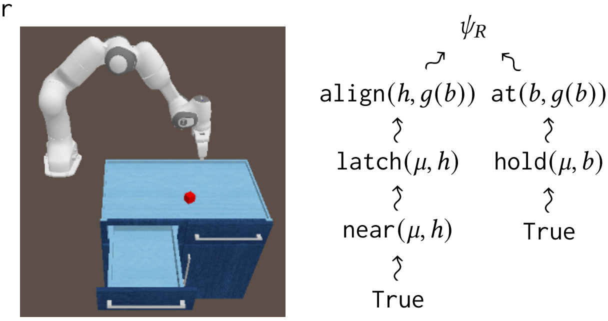


Fig. 6. Abstract Subtask Tree for PlaceCubeDrawer.

Program Learning. From an abstract environment defined by an abstract subtask tree T, RoboScribe synthesizes a robot-control program $\mathcal{P}$. First, we note that $\mathcal{P}$ can be derived by recursively traversing T and chaining the subtasks. Each abstract action is grounded as a neural network controller π that uses the attributes of the involved objects as input to generate low-level robot actions. In $\mathcal{P}$, each controller π runs until its corresponding subgoal condition is met. We depict the synthesized program for pick&place in Fig. 7 where the shorthand notations in the program represent loops:

```
def P_Pick&place(μ : EE):
  b := get(λb : block. true);
  π_grasp(μ, b) [hold(μ, b)];
  π_position(μ, b, g(b)) [at(b, g(b))]
```

Fig. 7. Synthesized program for pick&place by RoboScribe.

$$\pi(\mu, \{v\}, \{g(v)\})[\varphi] \equiv \texttt{while not}(\varphi) \ \{ \ \pi(\mu, \{v\}, \{g(v)\}) \ \}$$

Second, to learn each low-level controller π_φ in $\mathcal{P}$ for reaching states that satisfy its subgoal condition φ, we iteratively execute $\mathcal{P}$ and store trajectories from π_φ in a replay buffer B_φ. During gradient updates, RoboScribe trains π_φ by sampling from B_φ and optimizing it with an off-the-shelf off-policy RL algorithm. Paricularly, we use learned state abstraction predicates to provide dense reward signals for training low-level controllers. Dense reward functions are shaped systematically from predicates combined in arbitrary Boolean forms. For example, in Fig. 5, to train a controller for the subtask $\mathcal{T}_{2_1}$: $\texttt{hold}(g, b) \rightsquigarrow \texttt{at}(b, g(b))$, a dense reward function can be derived from the norm learned for $\texttt{at}(b, g(b))$, encouraging the robot to move its end effector closer to $g(b)$ with higher rewards for proximity.

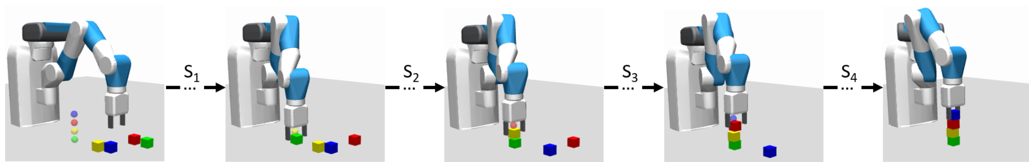


Fig. 8. Demonstration and partition for the Tower task.

2.4 Synthesizing Iterative Robot-Control Programs

While the synthesis strategy described in Sec.2.3 is applicable to multi-object tasks, it does not generalize well when scaling up to handle long-horizon tasks with varying numbers of objects. The abstraction method lacks the flexibility to capture the relationships and dependencies between an indefinite number of objects, particularly when coordination and sequencing are required. For example, in the `Tower` task shown in Fig.8, the goal is to synthesize a program $\mathcal{P}$ for a robot arm to stack a variable number of scattered blocks into a tower. Such tasks demand programs that can *iteratively* manage multiple instances of subtasks.

Key Challenge. Synthesizing iterative programs is challenging because it requires effective strategies for discovering repetitive subroutines and handling the complex dependencies between them. Unlike prior work (e.g. PROLEX [42] and Tabula [44]), RoboScribe does not assume predefined state and action abstraction and must be able to extract repetitive structures from demonstration trajectories within the robot's high-dimensional, continuous state and action spaces.

Key Insight. Our key idea is to leverage abstraction predicates learned *on-the-fly* to discover repetitive subroutines in demonstrations. During comparative abstraction refinement, when classifying states from task demonstrations and robot trajectories to learn new state abstraction predicates, RoboScribe identifies opportunities to reuse previously discovered predicates and their corresponding low-level controllers, as a means to uncover *abstract* repetitions within demonstrations.

Discovering Repetitive Subtasks. As shown in the learned abstraction for the `Pick&place` task in Eq. 2, RoboScribe can identify a state abstraction predicate $at(b, g(b))$ as a subgoal for placing one block b at its target and develop a routine of low-level controllers to achieve it. However, the task remains incomplete as additional blocks must still be placed. Using a single low-level controller for this would be insufficient as illustrated in Sec. 2.3. RoboScribe identifies that the predicate $at(b, g(b))$, interpreted with b as implicitly existentially quantified, can be reused to distinguish demonstration states where blocks are near their targets from unsuccessful single-policy attempts to complete the full task. It refines the abstraction in Eq.2 by defining a circular abstract subtask tree node to indicate this repetition:

$$\texttt{True} \rightsquigarrow \texttt{hold}(\mu, b) \rightsquigarrow at(b, g(b)) \rightsquigarrow \psi_R \tag{3}$$

This implies that the control strategy for achieving $\exists b.\ at(b, g(b))$ for some block b can be iteratively applied to handle remaining objects of the same type in the environment that have yet to meet this subgoal condition.

Learning Iterative Programs. RoboScribe synthesizes a loop structure for each circular abstract subtask tree node. This leads to an iterative `Tower` program $\mathcal{P}_{\texttt{Tower}}$ shown in Fig. 9, which intends to stack all the blocks on a table in a sequence, generated from the task abstraction in Eq. 3. However, determining the order in which blocks should be addressed within the sequence (among the loop iterations) remains unspecified. This ordering is particularly significant as placing a block in its goal position without first positioning the underlying blocks leads to failed outcomes. RoboScribe places a missing hole $??_p$ in the loop condition designated to specify an effective handling sequence in Fig. 9.

$$
\begin{array}{l}
\texttt{def}\ \mathcal{P}_{\text{Tower}}(\mu : \text{EE}):\\
\quad \texttt{while}\ \left((b := \mathbf{get}(\lambda b : \text{block}.\ ??_p))\right.\\
\qquad \left.\neq \mathit{null}\right):\\
\qquad \pi_{grasp}(\mu, b)\ \left[hold(\mu, b)\right]\\
\qquad \pi_{position}(b, g(b))\ \left[at(b, g(b))\right]
\end{array}
$$

Fig. 9. Iterative program $\mathcal{P}_{\text{Tower}}$ for Tower with a missing hole for the loop condition.

$$
\begin{array}{l}
\texttt{def}\ \mathcal{P}_{\text{Pick\&place}}(\mu : \text{EE}):\\
\quad \texttt{while}\ \left((b := \mathbf{get}(\lambda b : \text{block}.\ \neg at(b, g(b)) \wedge\right.\\
\qquad \neg\exists b'.g(b') \downarrow_z < g(b) \downarrow_z \wedge \neg at(b', g(b'))))\\
\qquad \left.\neq \mathit{null}\right):\\
\qquad \pi_{grasp}(\mu, b)\ \left[hold(\mu, b)\right]\\
\qquad \pi_{position}(b, g(b))\ \left[at(b, g(b))\right]
\end{array}
$$

Fig. 10. Synthesized iterative program $\mathcal{P}_{\text{Tower}}$ for Tower.

Loop Condition Synthesis. Conceptually, we can enumerate candidates to fill in the missing predicate $??_p$ based on the predicate production rules P defined in our DSL $\mathcal{L}$ (Fig. 4) and execute $\mathcal{P}_{\text{Tower}}$ in the real environment to empirically determine which predicate maximizes task performance, such as higher success rates. However, this approach is computationally prohibitive due to the extensive predicate search space of the DSL and the long-horizon nature of robot tasks that involve recurring objects. Additionally, training the low-level neural controllers in $\mathcal{P}_{\text{Tower}}$ depends on executing the program to obtain training data, resulting in a mutual dependency problem where loop condition synthesis and controller learning are interdependent. Our strategy circumvents these limitations by inferring the ordering predicate $??_p$ directly from demonstrations, avoiding the need for executing $\mathcal{P}_{\text{Tower}}$ in the real environment. Specifically, RoboScribe synthesizes an ordering predicate for $??_p$ that determines the correct sequence of object handling by analyzing the rationale in the demonstrations, explaining why certain objects are handled before others, such as why the red block is placed after the green and yellow blocks in Fig. 8. From demonstrations, RoboScribe learns such a predicate by enumerating predicates derivable from the production rules for P in the DSL $\mathcal{L}$ (see Fig. 4). In this process, we augment P with learned state abstraction predicates from abstract subtask trees as these predicates provide additional task-relevant constraints. We defer the formalization of the synthesis algorithm to Sec. 4.2.1. For `Tower`, RoboScribe synthesizes the following predicate for an effective handling sequence :

$$\neg at(b, g(b)) \wedge \neg\exists b'.g(b') \downarrow_z < g(b) \downarrow_z \wedge \neg at(b', g(b')) \tag{4}$$

which specifies that any block with a lower goal position must be placed before the current block. The termination condition ensures that once all blocks are in their goal positions, there are no further blocks to handle. The full program synthesized is given in Fig. 10.

3 Problem Setup

We study a learning paradigm where the agent can interact with many entities (objects) in an environment. The task for the agent is specified in the form of goals for the entities. We formalize it using the Entity-Factored Markov Decision Process (EFMDP) [65].

Throughout the paper, we use $\{v\}$ to denote a list. For a function f, we define element-wise application as $f(\{v\}) = \{f(v_1), \ldots, f(v_n)\}$.

Entity-Factored Markov Decision Process. An EFMDP with N entities is described by the tuple: $\mathcal{M} := \langle \Lambda, O = \{\mu, e_1, \ldots, e_N\}, \mathcal{S}, \mathcal{G}, \mathcal{A}, \mathbb{P}, \eta \rangle$. Here, Λ is a finite set of object types, e.g., cube and mug, and O is a finite set of objects, where μ and $\{e_1, \ldots, e_N\}$ are the agent (robot) and the entities, respectively. Each entity in O has a type drawn from Λ. Each object in O has an associated set of attributes drawn from a finite set $\mathcal{F} = \{f_1, f_2, \ldots, f_M\}$, for example, spatial coordinates $\{x, y, z\}$ in the 3D space. A state s in the state space $\mathcal{S}$ is a function $s : O \longrightarrow \mathbb{D}$ where $\mathbb{D}$ is the space of object descriptors, formally defined as $\mathbb{D} = (\mathcal{F} \longrightarrow \mathbb{R})$. This means that each object $o \in O$ is mapped to a function that assigns a real value to each attribute. For an object $o \in O$, $s(o)$ retrieves

$$\alpha ::= \mu \mid e_i \mid g(e_i) \mid \alpha\downarrow_{x,y,z} \mid \alpha\downarrow_{x,y} \mid \alpha\downarrow_{x} \mid \alpha\downarrow_{y} \mid \alpha\downarrow_{z} \mid \alpha - \alpha \mid \|\alpha\| \mid \mathtt{arctan2}(\alpha)$$
$$\psi ::= \alpha < \alpha \mid \alpha < \phi \mid \alpha > \phi \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \exists o.\, \psi$$

Fig. 11. Task Specifications of EFMDPs over objects $\mathcal{O} = \{\mu, e_1, \ldots, e_N\}$.

the object descriptor of o, i.e., $s(o) : \mathcal{F} \to \mathbb{R}$, and $s(o, \{f\})$ extracts the real values of the attributes $\{f\} \subseteq \mathcal{F}$, i.e., $s(o, \{f\}) = (s(o))(\{f\})$. We use $\mathrm{dom}(s)$ to retrieve the set of objects within a state s. We sometimes abuse notation for convenience to use $s(o)$ to refer to the full set of attribute values for o, i.e., the image of $s(o)$ under $\mathcal{F}$.

The ***goal space*** of an EFMDP $\mathcal{M}$ is denoted as $\mathcal{G}$. A goal command $g \in \mathcal{G}$ is a function (introduced in Sec. 2.1) $g : \mathcal{O} \to \mathbb{D}$ that defines the goal region for entities $\{e_1, \ldots, e_N\}$. Typically, $g(e_i)$ only maps a subset of e_i's attributes to a real value, specifying its desired placement. For instance, in Fig. 1, the goal regions (spheres) indicate the target positions for each block within the 3D space.

In an EFMDP $\mathcal{M}$, $\mathcal{A}$ is the robot's action space. The ***system dynamics*** of $\mathcal{M}$ is described by a probabilistic state transition function $\mathbb{P}(s'|s, a)$ for $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$, i.e., the robot's action can update the object states in its environment. The set of the initial states of an EFMDP is specified by $\eta : \mathcal{S} \to \mathbb{R}_{\geq 0}$ (i.e., $\eta(s)$ is the probability density of the initial state being s). A ***trajectory*** of an EFMDP $\zeta \in Z$ is a sequence $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$, where $s_i \in \mathcal{S}$ and $a_i \in \mathcal{A}$, where $s_{i+1} \sim \mathbb{P}(\cdot \mid s_i, a_i)$. EFMDPs can model several applications, including tabletop manipulation and scene reconfiguration. At the same time, the EFMDP contains more structure and symmetry compared to the standard MDP model, which can enable more efficient learning and better generalization [65].

Task Specification. We define predicates ψ used for robot task specifications of an EFMDP, as shown in Fig. 11, over the set of objects $\mathcal{O} = \{\mu, e_1, \ldots, e_N\}$ within. The operator $\downarrow$ extracts attribute values from $o \in \mathcal{O}$. The semantics of the predicates $[\![\psi]\!]$ are given in Fig. 12. Given a state s, we define $\varphi(s)$ as $[\![\varphi]\!](s)$, representing the truth value of the predicate φ in state s. We say that a trajectory $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots, s_H$ satisfies a task specification $\mathcal{T} : \varphi \rightsquigarrow \psi$, denoted as $\zeta \models \mathcal{T} : \varphi \rightsquigarrow \psi$, if $\psi(s_H)$ holds when $\varphi(s_0)$ holds. A task specification $\mathcal{T} : \mathtt{True} \rightsquigarrow \psi_R$ for an EFMDP $\mathcal{M}$ defines the intended behavior in the MDP. Starting from any possible initial EFMDP state $s_0 \sim \eta(\cdot)$, the agent is expected to reach a state s that satisfies $\psi_R(s)$. For example, for the Tower task in Fig. 1c involving the end effector μ and four blocks $\mathcal{O} = \{\mu, b_1, b_2, b_3, b_4\}$, its specification can be defined as:

$$\mathcal{T}_{\text{Tower}} : \mathtt{True} \rightsquigarrow \bigwedge_{i=1}^{4} \left\|b_i\downarrow_{x,y,z} - g(b_i)\right\| < \phi \wedge b_4\downarrow_z < \mu\downarrow_z$$

This specifies that all blocks must be placed in their goal regions, and the end effector must leave the top block. In this paper, we use ψ_R to denote the predicate encoding the task's final goal region, while ψ and φ typically represent intermediate subgoal conditions inferred by our algorithm. Predicates ψ defined over EFMDP objects $\mathcal{O}$ align with predicates P in the DSL $\mathcal{L}$, which are defined over program variables V binding EFMDP objects. This alignment allows inferred subgoal conditions to be lifted into program predicates. Thus, we use ψ and P interchangeably to refer to state abstraction predicates, based on the context.

Program Synthesis for Policy Learning. Given an EFMDP $\mathcal{M}$ with ***unknown*** state transition probabilities and task specification $\mathcal{T} : \mathtt{True} \rightsquigarrow \psi$, RoboScribe synthesizes a program $\mathcal{P}^*$ as a controller in the DSL $\mathcal{L}$ in Fig. 4 to fulfill $\mathcal{T}$. We outline the DSL operational semantics $\langle S, \rho, s\rangle \Downarrow (\rho', s')$ in Fig. 13. Formally, S is a program statement in Fig. 4, $\rho : V \to \mathcal{O}$ is an environment mapping that binds program variables V to objects in $\mathcal{O}$. For a program variable $v \in V$, at a given state s, $s(\rho(v))$ extracts the attribute values for the object referenced by v in s. Define $\mathtt{Exec}(\mathcal{M}, \mathcal{P})$ an interpreter that evaluates $\mathcal{P}$ in the EFMDP $\mathcal{M}$ based on the operational semantics and returns the EFMDP trajectory starting from a randomly sampled initial state $s_0 \sim \eta(\cdot)$. $\mathtt{Exec}$ terminates as

Expressions	**Predicates**
$[\![\alpha]\!](s) = \begin{cases} s(\mu), & \text{if } \alpha = \mu \\ s(e_i), & \text{if } \alpha = e_i \\ g(e_i), & \text{if } \alpha = g(e_i) \\ s([\![\alpha_1]\!](s), \{x,y,z\}), & \text{if } \alpha = \alpha_1 \downarrow_{x,y,z} \\ [\![\alpha_1]\!](s) - [\![\alpha_2]\!](s), & \text{if } \alpha = \alpha_1 - \alpha_2 \\ \lVert [\![\alpha_1]\!](s) \rVert, & \text{if } \alpha = \lVert \alpha_1 \rVert \\ \texttt{arctan2}([\![\alpha_1]\!](s)), & \text{if } \alpha = \texttt{arctan2}(\alpha_1) \end{cases}$	$[\![\psi]\!](s) = \begin{cases} [\![\alpha_1]\!](s) < [\![\alpha_2]\!](s), & \text{if } \psi = \alpha_1 < \alpha_2 \\ [\![\alpha]\!](s) < \phi, & \text{if } \psi = \alpha < \phi \\ [\![\alpha]\!](s) > \phi, & \text{if } \psi = \alpha > \phi \\ \neg[\![\psi_1]\!](s), & \text{if } \psi = \neg\psi_1 \\ [\![\psi_1]\!](s) \wedge [\![\psi_2]\!](s), & \text{if } \psi = \psi_1 \wedge \psi_2 \\ [\![\psi_1]\!](s) \vee [\![\psi_2]\!](s), & \text{if } \psi = \psi_1 \vee \psi_2 \\ \bigvee_i [\![\psi_1[o \mapsto e_i]]\!](s), & \text{if } \psi = \exists o.\psi_1 \end{cases}$

Fig. 12. Semantics of task specifications of EFMDPs over objects $O = \{\mu, e_1, \ldots, e_N\}$.

$$\frac{\langle S_1, \rho, s\rangle \Downarrow (\rho', s') \quad \langle S_2, \rho', s'\rangle \Downarrow (\rho'', s'')}{\langle S_1; S_2, \rho, s\rangle \Downarrow (\rho'', s'')} \qquad \frac{\langle P, \rho, s\rangle \Downarrow \text{true} \quad \langle S_1, \rho, s\rangle \Downarrow (\rho', s')}{\langle \textbf{if}(P)\ S_1\ \textbf{else}\ S_2, \rho, s\rangle \Downarrow (\rho', s')} \qquad \frac{\langle P, \rho, s\rangle \Downarrow \text{false} \quad \langle S_2, \rho, s\rangle \Downarrow (\rho', s')}{\langle \textbf{if}(P)\ S_1\ \textbf{else}\ S_2, \rho, s\rangle \Downarrow (\rho', s')}$$

$$\frac{\langle P, \rho, s\rangle \Downarrow \text{true} \qquad \langle S, \rho, s\rangle \Downarrow (\rho', s') \qquad \langle \textbf{while}(P)\ \{S\}, \rho', s'\rangle \Downarrow (\rho'', s'')}{\langle \textbf{while}(P)\ \{S\}, \rho, s\rangle \Downarrow (\rho'', s'')} \qquad \frac{\langle P, \rho, s\rangle \Downarrow \text{false}}{\langle \textbf{while}(P)\ \{S\}, \rho, s\rangle \Downarrow (\rho, s)}$$

$$\frac{o : \tau \in \text{dom}(s) \qquad \langle P, \rho[v \mapsto o], s\rangle \Downarrow \text{true}}{\langle v := \textbf{get}(\lambda v : \tau.\ P), \rho, s\rangle \Downarrow (\rho[v \mapsto o], s)} \qquad \frac{\forall o : \tau \in \text{dom}(s).\ \langle P, \rho[v \mapsto o], s\rangle \Downarrow \text{false}}{\langle v := \textbf{get}(\lambda v : \tau.\ P), \rho, s\rangle \Downarrow (\rho[v \mapsto \mathit{null}], s)}$$

$$\frac{a \sim \pi_\theta\big(s(\rho(\mu)),\ s(\rho(\{v\})),\ g(\rho(\{v'\}))\big) \qquad s' \sim \mathbb{P}(\cdot \mid s, a)}{\langle \pi_\theta(\mu,\ \{v\},\ g(\{v'\})), \rho, s\rangle \Downarrow (\rho, s')}$$

$$\frac{\langle \alpha, \rho, s\rangle \Downarrow u}{\langle \alpha < \phi, \rho, s\rangle \Downarrow u < \phi} \qquad \frac{\langle \alpha, \rho, s\rangle \Downarrow u}{\langle \alpha \neq \mathit{null}, \rho, s\rangle \Downarrow u \neq \mathit{null}} \qquad \frac{\langle P_1, \rho, s\rangle \Downarrow b_1 \quad \langle P_2, \rho, s\rangle \Downarrow b_2}{\langle P_1 \wedge P_2, \rho, s\rangle \Downarrow b_1 \wedge b_2} \qquad \frac{\langle P, \rho, s\rangle \Downarrow b}{\langle \neg P, \rho, s\rangle \Downarrow \neg b}$$

$$\frac{o : \tau \in \text{dom}(s) \qquad \langle P, \rho[v \mapsto o], s\rangle \Downarrow \text{true}}{\langle \exists v : \tau.\ P, \rho, s\rangle \Downarrow \text{true}} \qquad \frac{\forall o : \tau \in \text{dom}(s).\ \langle P, \rho[v \mapsto o], s\rangle \Downarrow \text{false}}{\langle \exists v : \tau.\ P, \rho, s\rangle \Downarrow \text{false}} \qquad \frac{\rho(v) \in \text{dom(s)}}{\langle v, \rho, s\rangle \Downarrow \rho(v)}$$

$$\frac{\rho(v) \in \text{dom(s)}}{\langle g(v), \rho, s\rangle \Downarrow g(\rho(v))} \qquad \frac{\langle \alpha, \rho, s\rangle \Downarrow o \qquad o \in \text{dom}(s)}{\langle \alpha \downarrow_{x,y,z}, \rho, s\rangle \Downarrow s(o, \{x,y,z\})} \qquad \frac{\langle \alpha_1, \rho, s\rangle \Downarrow u_1 \qquad \langle \alpha_2, \rho, s\rangle \Downarrow u_2}{\langle \alpha_1 - \alpha_2, \rho, s\rangle \Downarrow u_1 - u_2}$$

Fig. 13. The DSL $\mathcal{L}$ operational semantics in RoboScribe.

soon as a specification-satisfying state is encountered. The learning objective is to synthesize $\mathcal{P}^*$:

$$\mathcal{P}^* = \arg\max_{\mathcal{P} \in \mathcal{L}} \Pr_{\zeta \sim \texttt{Exec}(\mathcal{M}, \mathcal{P})} [\zeta \models \mathcal{T} : \texttt{True} \rightsquigarrow \psi_R] \tag{5}$$

In practice, to evaluate task success, we define $\mathcal{P}^* \models_{\mathcal{M},\epsilon} \texttt{True} \rightsquigarrow \psi_R$ meaning that finite-length trajectories *sampled* from $\texttt{Exec}(\mathcal{M},\ \mathcal{P})$ *empirically* satisfy the goal condition ψ_R with a probability of at least $1 - \epsilon$. The values for the maximum trajectory length and ϵ are user-configurable.

4 Abstraction Refinement-guided Robot Control Program Synthesis

We present the core algorithms for the abstraction refinement-guided synthesis strategy in RoboScribe. We first provide the top-level synthesis algorithm, and then describe its key components.

4.1 Top-level Algorithm

Algorithm 1 The RoboScribe Procedure

1: **procedure** RoboScribe($\mathcal{M}$, $\mathcal{L}$, $\mathcal{D}$, $\mathtt{True} \rightsquigarrow \psi_R$)
2: $\quad T \leftarrow (N = \{u_{\mathtt{True}}, u_{\psi_R}\},\ E = \{u_{\mathtt{True}} \rightarrow u_{\psi_R}\},\ \psi_R)$
3: $\quad \mathcal{P}^*, T^* \leftarrow$ synthesize($\mathcal{M}$, $\mathcal{L}$, $\mathcal{D}$, T, ψ_R)
4: $\quad$ **return** $\mathcal{P}^*$

The top-level RoboScribe algorithm is presented in Algorithm 1. It takes as input an EFMDP $\mathcal{M} = \{\Lambda, O, \mathcal{S}, \mathcal{G}, \mathcal{A}, \mathbb{P}, \eta\}$, the DSL $\mathcal{L}$ (defined in Fig. 4), a set of task demonstrations $\mathcal{D}$, and a task specification $\mathtt{True} \rightsquigarrow \psi_R$. ***The DSL $\mathcal{L}$ does not predefine useful state and action abstractions, which are yet to be learned as part of the synthesis process***. The objective is to synthesize a program $\mathcal{P}^*$ in $\mathcal{L}$ that satisfies the specification $\mathtt{True} \rightsquigarrow \psi_R$ (Eq. 5). As in conventional RL settings, the goal condition ψ_R is unknown to RoboScribe. However, the agent can use ψ_R as a black box to query whether any state encountered s satisfies ψ_R.

Abstract Subtask Trees. During its synthesis procedure, RoboScribe maintains state and action abstraction of a robot environment as an abstract subtask tree T—a hierarchical representation that encodes the sequence and relationships among subtasks for reaching the task's goal states.

Definition 4.1 (Abstract Subtask Tree). An Abstract Subtask Tree $T = (N, E, \psi_R)$ is a tuple:

- N is a set of nodes, each representing a state abstraction predicate, denoted by φ or ψ, which defines a subset of the EFMDP state space. Throughout the paper, we use the terms predicate φ and tree node u_φ interchangeably.
- $E \subseteq N \times N$ is a set of directed edges between nodes, with each edge $u_\varphi \rightarrow u_\psi \in E$ representing a subtask $\varphi \rightsquigarrow \psi$ of the overall task, which transitions the agent from states characterized by φ to states in ψ, In the following, we also use edge $u_\varphi \rightarrow u_\psi$ and subtask $\varphi \rightsquigarrow \psi$ interchangeably.
- $\psi_R \in N$ is the root node, encapsulating the goal states of the overall task.

In an abstract subtask tree, state abstraction predicates on the tree nodes serve as decomposition of a complex robotic task. Tree edges represent abstract actions to transition between key subgoals. A tree path ($\mathtt{True} \rightsquigarrow \varphi_1 \rightsquigarrow \varphi_2 \rightsquigarrow \ldots \rightsquigarrow \psi_R$) in T, leading toward the goal states at the root, is a sequence of subtasks, guiding the agent from one subtask to the next until the whole task is complete. The tree is *inverted*, for any node with multiple predecessors, the agent is directed to execute subtasks associated with each predecessor node, recursively. Fig. 6 displays the abstract subtask tree for a multi-object environment.

Initial Environment Abstraction. In Algorithm 1, at line 2, RoboScribe creates the initial environment abstraction as an abstract subtask tree T with two nodes $u_{\mathtt{True}}$ and u_{ψ_R} corresponding to the set of all possible initial states and environment states that satisfy the unknown goal condition. The edge $u_{\mathtt{True}} \rightarrow u_{\psi_R}$ represents a controller that satisfies the task specification $\mathtt{True} \rightsquigarrow \psi_R$. At line 3, Algorithm 1 invokes the Synthesize procedure (detailed in Algorithm 2) to iteratively refines the coarse initial abstraction by need into a hierarchy of subtasks, continuing until a valid task-solving program is obtained within the abstracted environment.

4.2 The Main Synthesis Procedure

Synthesis Rules. We describe the Synthesize procedure using the synthesis rules of the following shape:

$$\mathcal{M}, \mathcal{L}, \mathcal{D}, T \vdash (\mathcal{P}^*, T^*) : \psi$$

where $\mathcal{M}$, $\mathcal{L}$, $\mathcal{D}$ are the task EFMDP, our DSL (Fig. 4), and the task demonstrations respectively. T is an initial abstract subtask tree. The rule specifies the refinement of T into a valid abstraction T^*, which can then be converted into a program $\mathcal{P}^*$ whose execution fulfills the goal condition ψ. Fig. 14 depicts the synthesis rules. Both rules rely on a procedure $\mathcal{M}, \mathcal{D}, T \triangleright \mathcal{P}$ that synthesizes a

Policy

$$\frac{\mathcal{M},\mathcal{D},T \triangleright \mathcal{P} \qquad \mathcal{P} \models_{\mathcal{M},\epsilon} \texttt{True} \rightsquigarrow \psi}{\mathcal{M},\mathcal{L},\mathcal{D},T \vdash\ (\mathcal{P},T):\psi}$$

Refinement

$$\frac{\begin{array}{c}\mathcal{M},\mathcal{D},T \triangleright \mathcal{P} \qquad \mathcal{P} \not\models_{\mathcal{M},\epsilon} \texttt{True} \rightsquigarrow \psi \qquad P_s \leftarrow \{s_t \mid s_t \not\models \psi \wedge s_{t+1} \models \psi \wedge \{s_t, s_{t+1}\} \in \mathcal{D}\} \\ N_s \leftarrow \{\zeta \sim \texttt{Exec}(\mathcal{M},\mathcal{P})\} \qquad \varphi \in \mathcal{L} \wedge \varphi(s) = 1 \text{ for } s \in P_s \wedge \varphi(s) = 0 \text{ for } s \in N_s \\ \mathcal{M},\mathcal{L},\mathcal{D},T[\texttt{True} \overset{\varphi}{\rightsquigarrow} \psi] \vdash (\mathcal{P}',T'):\varphi \qquad \mathcal{M},\mathcal{L},\mathcal{D},T' \vdash (\mathcal{P}'',T''):\psi\end{array}}{\mathcal{M},\mathcal{L},\mathcal{D},T \vdash\ (\mathcal{P}'',T''):\psi}$$

Fig. 14. The RoboScribe Synthesis Procedure

Algorithm 2 $\mathcal{M}$, $\mathcal{L}$, $\mathcal{D}, T \vdash\ (\mathcal{P}'', T'') : \psi$ The Main Synthesis Procedure

1: **procedure** Synthesize($\mathcal{M}$, $\mathcal{L}$, D, T, ψ)
2: $\quad$ $\mathcal{P} \leftarrow$ GenProgram($\mathcal{M}$, $\mathcal{D}$, T)
3: $\quad$ **if** $\mathcal{P} \models_{\mathcal{M},\epsilon} \texttt{True} \rightsquigarrow \psi$ **then**
4: $\quad\quad$ **return** $\mathcal{P}$, T
5: $\quad$ **else**
6: $\quad\quad$ $P_s \leftarrow \{s_t \mid s_t \not\models \psi \wedge s_{t+1} \models \psi \wedge \{s_t, s_{t+1}\} \in \mathcal{D}\}$
7: $\quad\quad$ $N_s \leftarrow \{\zeta \sim \texttt{Exec}(\mathcal{M},\mathcal{P})\}$
8: $\quad\quad$ $\varphi \leftarrow$ `LearnClassifier`(P_s, N_s, $\mathcal{L}$, T, ψ)
9: $\quad\quad$ $T \leftarrow$ UpdateTree(T, [$\texttt{True} \overset{\varphi}{\rightsquigarrow} \psi$])
10: $\quad\quad$ $\mathcal{P}', T' \leftarrow$ Synthesize($\mathcal{M}$, $\mathcal{L}$, D, T, φ)
11: $\quad\quad$ $\mathcal{P}'', T'' \leftarrow$ Synthesize($\mathcal{M}$, $\mathcal{L}$, D, T', ψ)
12: $\quad\quad$ **return** $\mathcal{P}''$, T''

program $\mathcal{P}$ from the abstracted environment T and ground the abstract actions in $\mathcal{P}$ as low-level neural controllers in the real environment $\mathcal{M}$. We defer the discussion of this procedure to Sec. 4.3.

The Policy rule applies when the program $\mathcal{P}$ derived from the abstract subtask tree T can directly satisfy the specification, i.e., $\mathcal{P} \models_{\mathcal{M},\epsilon} \texttt{True} \rightsquigarrow \psi$ (we set $1-\epsilon$ as a lower bound for the probability of task success). In this case, the rule directly outputs ($\mathcal{P}$, T) as the synthesized solution. The Refinement rule, on the other hand, addresses cases where the program $\mathcal{P}$ generated from T does not fully solve the task. Here, our key idea is to systematically compare states within successful task demonstrations with those from failed behaviors by the program $\mathcal{P}$ executed in the real environment, aiming to identify pivotal states that are essential for enabling task success. As stated in the Refinement rule, RoboScribe extracts states just before task success from demonstrations, forming a set $P_s = \{s_t \mid s_t \not\models \psi \wedge s_{t+1} \models \psi\}$, and compares them with states N_s from the learned controller's trajectories to learn what prerequisite conditions are necessary. RoboScribe synthesizes a state abstraction predicate φ' to distinguish between P_s and N_s, refining the abstraction T by breaking down the task $\texttt{True} \rightsquigarrow \psi$, which results in a new abstract subtask tree $T[\texttt{True} \overset{\varphi}{\rightsquigarrow} \psi]$ that adds φ as an intermediate subgoal for ψ in T (formalized in Sec. 4.2.2). We hypothesize that having learned how to achieve φ by the synthesized program $\mathcal{P}'$ from φ, it is an easier task for the agent to learn a program $\mathcal{P}''$ based on $\mathcal{P}'$ to achieve the goal condition ψ. Notably, the Refinement rule embodies a recursive task decomposition process to repeatedly refine an initially coarse abstraction until a valid task-solving program can be obtained. Algorithm 2 operationalizes the synthesis rules in a recursive function Synthesize. At line 8 and line 9, the Synthesize function

Algorithm 3 $\varphi \in \mathcal{L}$ s.t. $\varphi(s) = 1$ for $s \in P_s$, $\varphi(s) = 0$ for $s \in N_s$: Learn a classifier for P_s and N_s

1: **procedure** LearnClassifier(P_s, N_s, $\mathcal{L}$, T, ψ)
2: **if** $\exists \varphi.\ \varphi \rightsquigarrow \psi \in T\ \bigwedge\ \forall s \in P_s.\ \text{ExistQuant}(\varphi)(s) \bigwedge \forall s \in N_s.\ \neg(\text{ExistQuant}(\varphi))(s)$ **then**
3: **return** φ
4: **else**
5: $\text{Exp}_\alpha \leftarrow \mathcal{L}(\alpha)$
6: $\varphi \leftarrow$ LearnDecisionTree(Exp_α, P_s, N_s)
7: **return** φ

invokes LearnClassifier and UpdateTree for state abstraction refinement. We formalize these two procedures below, starting with key notations.

Given a predicate ψ over EFMDP objects $O = \{\mu, e_1, \ldots, e_N\}$ (defined in Fig. 11), let $\text{Entities}(\psi) \subseteq \{e_1, \ldots, e_N\}$ denote the set of entity variables that appear free in ψ. We define the transformation:

$$\text{ExistQuant}(\psi) := \exists e \in \text{Entities}(\psi).\ \psi$$

which lifts entity variables $e_1, \ldots, e_N$ appearing in ψ free existentially quantified.

4.2.1 Learning State Abstraction Predicates

We illustrate the state abstraction predicate learning procedure LearnClassifier in Algorithm 3. It takes as input P_s the set of states prior to the states that satisfy a goal or subgoal condition ψ in the demonstration $\mathcal{D}$, N_s the set of states along the agent's behavior that failed to reach ψ, the DSL $\mathcal{L}$, the abstract subtask tree T representing the current environment abstraction, and ψ, aiming to learn a state abstraction predicate capturing what should have been achieved by the agent in order to enable reaching states in ψ. At line 2, the algorithm checks whether the state abstraction predicate φ from any existing predecessor of ψ can be reused to distinguish P_s and N_s through $\text{ExQuant}(\psi)$. For example, in the Tower task described in Sec. 2.4, once the state abstraction predicate $at(b, g(b))$ is identified as a subgoal for placing a block b at its target and a corresponding subroutine of controllers is learned, the task remains incomplete as additional blocks still require positioning. The existentially quantified predicate $\exists b.\ at(b, g(b))$ helps distinguish demonstration states where blocks are correctly positioned near their targets from failed attempts by a single controller struggling to complete the task.

If reusing an existing predicate is not possible, in Algorithm 3, LearnClassifier synthesizes a decision tree (DT) at line 6 to separate P_s and N_s using features from expressions α derived from the production rules in our task specification language in Fig. 11. The hypothesis set of LearnDecisionTree consists of Boolean combinations of predicates of the form $\alpha \leq \phi$, with ϕ being a constant threshold, which are learned during training. Standard DT learning algorithms begin with an empty tree, greedily selecting features that maximize information gain, and continue until all leaves are labeled with a single class. Finally, the learned DT is converted to a predicate φ.

4.2.2 Refining Abstract Subtask Trees

With the learned classifier φ as a prerequisite for achieving ψ in a task $\texttt{True} \rightsquigarrow \psi$, UpdateTree refines the abstract subtask tree T by using φ as a subgoal to decompose the task. We specify this refinement procedure $T[\texttt{True} \overset{\varphi}{\rightsquigarrow} \psi]$ in Algorithm 4. At line 2, if the newly identified subtask goal predicate φ matches an existing subtask goal φ' that precedes ψ in T (i.e., the LearnClassifier procedure has opted to reuse φ'), a repeating subroutine is effectively recognized. This means that the controllers designed to achieve φ' for manipulating some objects can be repurposed to solve the subtask to reach φ for a different set of objects of the same type. The algorithm marks $u_{\varphi'}$ as a circular node φ' at line 4 to indicate this repetition - the control strategy used to reach φ' should

Algorithm 4 $T[\texttt{True} \overset{\varphi}{\rightsquigarrow} \psi]$: Update an abstract subtask tree $T = (N, E, \psi_R)$

1: **procedure** UPDATETREE(T, $[\texttt{True} \overset{\varphi}{\rightsquigarrow} \psi]$)
2:     **if** $\exists\varphi'.\ \varphi' \rightsquigarrow \psi \in T \wedge \varphi' \equiv \varphi$ **then**
3:         $\varphi' \leftarrow \text{ExistQuant}(\varphi')$
4:         $u_{\varphi'} \leftarrow \varphi'$
5:     **else if** $\exists\varphi'.\ \varphi' \rightsquigarrow \psi \in T \wedge$ ENTITIES(φ') = ENTITIES(φ) **then**
6:         $N,\ E \leftarrow N \cup \{u_\varphi\},\ E \setminus \{u_{\varphi'} \to u_\psi\} \cup \{u_{\varphi'} \to u_\varphi,\ u_\varphi \to u_\psi\}$
7:     **else**
8:         $N,\ E \leftarrow N \cup \{u_\varphi\},\ E \cup \{u_{\texttt{True}} \to u_\varphi, u_\varphi \to u_\psi\}$

then be executed iteratively to address recurring objects in the environment that have not yet satisfied φ'. For this purpose, we have made φ' existentially quantified.

An invariant we maintain for an abstract subtask tree T is that for each tree node ψ, for any predecessors $\varphi \rightsquigarrow \psi$ and $\varphi' \rightsquigarrow \psi$, φ and φ' are subgoal conditions for unique sets of objects, and hence requiring different control strategies (i.e. unique tree paths towards φ and φ') and otherwise they should be collapsed into a circular node. For example, consider the `PlaceCubeDrawer` task depicted in Fig.6, which has an abstract subtask tree consisting of two main paths: one for opening the drawer and the other for placing the cube inside it. During the refinement of $T[\texttt{True} \overset{\varphi}{\rightsquigarrow} \psi]$, if a newly identified subgoal φ targets the same set of entities as an existing predecessor φ' of ψ (i.e. Entities(φ) = Entities(φ')), the UPDATETREE procedure in Algorithm4 inserts a new node $u_{\varphi'}$ between u_φ and u_ψ at line 6, establishing φ' as an intermediate subgoal for $\varphi \rightsquigarrow \psi$. For example, in the tree refinement illustrated by Equation 2, the predicate $\texttt{at}(b, g(b))$, which signifies that block b is at its goal, is added between the predicate $\texttt{hold}(\mu, b)$—indicating the robot is gripping b—and ψ_R, the overall task's goal condition. If φ involves different set of entities from any existing predecessor φ' of ψ, a distinct tree path from u_φ to u_ψ is created at line 8, representing distinct objects to control, as exemplified in the unique two paths in the abstract subtask tree for `PlaceCubeDrawer` in Fig. 6.

4.3 Synthesizing Robot-Control Programs from Abstract Subtask Trees

An important step in RoboScribe is synthesizing an executable program $\mathcal{P}$ from an abstract subtask tree T. We formalize this procedure as $\mathcal{M}, \mathcal{D}, T \triangleright \mathcal{P}$, implemented in a procedure GENPROGRAM described in Algorithm 5. Our main synthesis procedure SYNTHESIZE invokes GENPROGRAM in Algorithm 2 (line 2) to generate a candidate program and does so repeatedly for each *refined* abstract subtask tree until a specification-satisfying program can be synthesized.

The GENPROGRAM procedure $\mathcal{M}, \mathcal{D}, T \triangleright \mathcal{P}$ accomplishes three main objectives as formalized in Algorithm 5: (1) it constructs the "skeletion" of $\mathcal{P}$ based on the hierarchical structure of T via the TREE2PROGRAM procedure at line 2. (2) TREE2PROGRAM also infers loops within $\mathcal{P}$ to handle varying numbers of objects by detecting repeated patterns in the demonstration $\mathcal{D}$. (3) It grounds the abstract actions in $\mathcal{P}$ as low-level controllers that can solve the subtasks within T through the TRAINPROGRAM procedure at line 5. These low-level controllers are neural network policies that operate directly in the robot environment to control robot actions.

4.3.1 Program Generation

The TREE2PROGRAM procedure (Line 8 of Algorithm 5) traverses an abstract subtask tree T rooted at u_φ. It generates a program P_{u_φ} from T for solving the task of reaching states satisfying φ. Here we assume that in a multi-object setting, task demonstrations implicitly indicate the order in which multiple objects of different types should be handled (we relax this assumption in Sec. 4.4). For

Algorithm 5 $\mathcal{M}, \mathcal{D}, T \triangleright \mathcal{P}$: Synthesize a program $\mathcal{P}$ from an abstract subtask tree $T = (N, E, \psi_R)$

1: **procedure** GenProgram($\mathcal{M}$, $\mathcal{D}$, T)
2: $\quad\mathcal{P} \leftarrow$ Tree2Program(ψ_R, $\mathcal{D}$, T)
3: $\quad$**for all** $o \in \text{FreeVars}(\mathcal{P})$: **do**
4: $\quad\quad\mathcal{P} \leftarrow \{v := \mathbf{get}(\lambda v : \Lambda(o).\ \text{True})\};\ \mathcal{P}\big|_{o \mapsto v}$ $\qquad\triangleright$ v fresh
5: $\quad\mathcal{P}^* \leftarrow$ TrainProgram($\mathcal{M}$, $\mathcal{P}$)
6: $\quad$**return** $\mathcal{P}^*$
7:
8: **procedure** Tree2Program(u_φ, $\mathcal{D}$, T)
9: $\quad\mathcal{P}_{u_\varphi} \leftarrow \{\}$
10: $\quad$**for all** edge $(e \equiv u_{\varphi'} \longrightarrow u_\varphi) \in T$ sorted by $i(\mathcal{D}, \varphi')$ **do**
11: $\quad\quad\mathcal{P}_{u_{\varphi'}} \leftarrow$ Tree2Program($u_{\varphi'}$, $\mathcal{D}$, T)
12: $\quad\quad\mathcal{P}_{u_\varphi} \leftarrow \mathcal{P}_{u_\varphi};\ \mathcal{P}_{u_{\varphi'}}$
13: $\quad$**if** $u_\varphi \equiv \exists o.\ \varphi$ **then** $\qquad\triangleright$ u_φ induces an iterative procedure
14: $\quad\quad v \leftarrow$ FreshVar()
15: $\quad\quad p \leftarrow$ PredicateSynthesis($\mathcal{D}$, v, $\exists o.\ \varphi$) $\qquad\triangleright$ Loop Condition Synthesis
16: $\quad\quad\mathcal{P}_{u_\varphi} \leftarrow \mathbf{while}(v := \mathbf{get}(\lambda v.\ p))\ \{(\mathcal{P}_{u_\varphi};\ \pi_\varphi[\varphi])\big|_{o \mapsto v}\};$
17: $\quad$**else**
18: $\quad\quad\mathcal{P}_{u_\varphi} \leftarrow \mathcal{P}_{u_\varphi};\ \pi_\varphi[\varphi]$
19: $\quad$**return** $\mathcal{P}_{u_\varphi}$

example, in the PlaceCubeDrawer task shown in Fig. 6, the agent must first pull the drawer open using its handle before placing the cubes inside. Define $i(\mathcal{D}, \varphi')$ as the position in the demonstration where φ' holds. Tree2Program enumerates the incoming edges $u_{\varphi'} \longrightarrow u_\varphi$ of u_φ (line 10) in the order of $i(\mathcal{D}, \varphi')$, recursively applying itself to $u_{\varphi'}$ (line 11), and appending the resulting program $P_{u_{\varphi'}}$ to P_{u_φ} (line 12). At line 18, the algorithm appends a low-level controller $\pi_\varphi = \pi_\theta(\mu, \{o\}, \{g(o')\})$, a neural network policy with trainable weights θ, to the program P_{u_φ}. This controller guides the agent from states satisfying the subgoals in the predecessors of u_φ (namely $\{\varphi'\}$) to states that satisfy φ in u_φ. Here, $\{o\} = \text{Entities}(\varphi)$ denotes the set of entities involved in φ, while $\{g(o')\}$ represents the goal conditions in φ for these entities, with o' potentially being a subset of o. The controller π_φ needs to manage the entities in o to achieve the subgoal condition φ. If u_φ is designated as a circular node, as constructed in Sec. 4.2.2, the subroutine synthesized in $\mathcal{P}u_\varphi$ is designed for repeated execution to handle recurring objects of the same types to achieve φ in a loop. To simplify the presentation, we assume a single existential quantifier for the subgoal condition related to u_φ, though the algorithm trivially extends to multiple quantifiers. We introduce a fresh program variable v to bind recurring objects within the loop. At line 16, Tree2Program constructs a loop with $(\mathcal{P}u_\varphi; \pi_\varphi\big|_{o \mapsto v})$ as its body, replacing o in the program with v to track recurring objects bound in each iteration, analogous to existential quantifier instantiation. Fig. 9 illustrates this process for the Tower task. We note that at Line 4 of Algorithm 5, for any remaining object identifier o in a synthesized program $\mathcal{P}$, we similarly project o to a fresh variable v and prepend $v := \mathbf{get}(\lambda v : \Lambda(o).\text{True})$ to $\mathcal{P}$. This allows the program to generalize across environments by retrieving the appropriate entity, removing dependencies on specific object identifiers (see Fig. 7 as an example).

So far, the algorithm does not specify the order in which objects are addressed in the loop iterations, which is crucial for tasks with dependencies, such as `Tower` (Fig. 8) where placing a

Algorithm 6 Learning Loop Conditions from Demonstrations

1: **procedure** PredicateSynthesis($\mathcal{D}, v, \varphi$)
2:     **for** $1 \leq i \leq N$ **do**
3:         $P_i, N_i \leftarrow \{\}, \{\}$
4:     **for** each demonstration rollout $d \in \mathcal{D}$ **do**
5:         Let $e_1, \ldots, e_N$ be the objects sorted by the order handled by φ and
6:         S_i be the partition of d in which e_i is handled
7:         **for all** e_i **do**
8:             $P_i \leftarrow P_i \cup \{(s_t, e_i) \mid s_t \in S_i\}$
9:             **for all** S_j s.t. $j < i$ **do**
10:                 $N_i \leftarrow N_i \cup \{(s_t, e_i) \mid s_t \in S_j\}$
11:             **for all** S_j s.t. $j > i$ **do**
12:                 $N_i \leftarrow N_i \cup \{(s_t, e_i) \mid s_t \in S_j\}$
13:     $P \leftarrow True$
14:     **for** $1 \leq i \leq N$ **do**
15:         $\psi \leftarrow$ TopdownEnum(P_i, N_i)
16:         $P \leftarrow P \wedge \psi[e_i \mapsto v]$
17:     **return** P

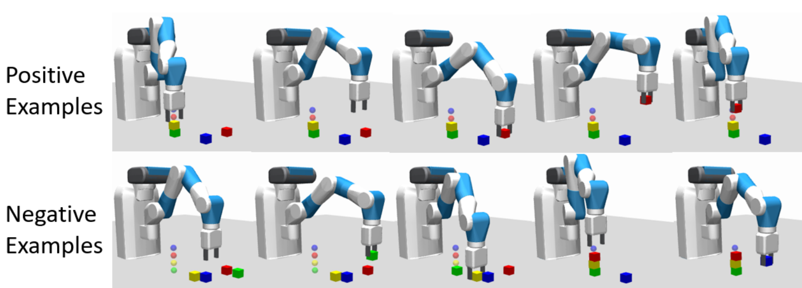


Fig. 15. Positive and negative examples for learning the predicate for when to handle the red block in `Tower`. In the demonstration, the agent stacks green, yellow, red and blue blocks in order.

block in its goal position before positioning the underlying blocks can result in failure. To this end, Tree2Program invokes PredicateSynthesis in Algorithm 6 to specify an effective handling sequence for the loop structure. We outline this procedure as follows.

4.3.2 Loop Inference

Our key approach to identifying the potential order for an effective handling sequence for recurring objects is analyzing the underlying rationale in task demonstrations to understand why certain objects must be handled before others.

Given the set of demonstrations $\mathcal{D}$ and a circular tree node φ in an abstract subtask tree T, the PredicateSynthesis procedure in Algorithm 6 formalizes the generation of loop conditions for synthesized iterative programs. Starting from line 5, for each rollout d from the demonstrations D, we sort objects $e_1, e_2, \ldots$ of the desired types in d according to their order of satisfying φ in d and partition d based on this order. Each partition S_i represents a subtask period where e_i is handled in the demonstrations. For example, for the abstraction of `Tower` in Eq. 3, given the demonstration in Fig. 8, we instantiate the existential quantifiers b in the circular node $\exists b.\ at(b, g(b))$ with the colored blocks respectively. The agent stacks the green, yellow, red, and blue blocks sequentially from bottom to top, resulting in partitions S_1, S_2, S_3 and S_4 in Fig. 8.

For each e_i, RoboScribe maintains positive examples (s_t, e_i) for all states s_t in S_i where e_i is handled at line 8, and negative examples (s_t, e_i) for all states s_t in prior $(S_0, \ldots, S_{i-1})$ and subsequent $(S_{i+1}, \ldots)$ partitions to illustrate why e_i should not be handled earlier at line 10 or does not need to be handled afterwards at line 12 in these negative states. For instance, in Fig. 15, RoboScribe shows why the red block is placed after the green and yellow blocks. In the top row, where the green and yellow blocks are already positioned, any state involving the placement of the red block is considered a positive example. In the bottom row, where the green and yellow blocks are not yet in their goal positions, the placement of the red block is marked as a negative example (it should not be handled), and any subsequent states after the red block is positioned are also negative examples (since its handling is already complete). Given the positive and negative examples $\{(s_t, e_i)\}^+, \{(s_t, e_i)\}^-$, any classifier that defines the relationship between e_i and other objects in s_t (abstracted as existential variables) and effectively separates the examples, provides both an ordering and a termination constraint for handling e_i during manipulation.

At line 5, RoboScribe uses top-down synthesis to generate a classifier predicate for each e_i, following the production rules for predicates ψ in our task specification language (see Fig. 11). In this process, we augment ψ with learned state abstraction predicates from abstract subtask trees as these predicates provide additional task-relevant constraints. The learned predicates for all e_i are combined to fill in the loop condition in the synthesized program. Here we use top-down enumeration instead of decision tree (DT) learning because loop conditions for handling sequences typically require existential quantifiers to manage unbounded entities with dependencies, which are not well-suited for DTs. If the synthesis algorithm does not find a classifier predicate for an object e_i within a reasonable search budget, RoboScribe interprets this as the absence of an ordering constraint for that e_i — the agent can select e_i for manipulation without restrictions, and it then returns true in this case. Given the examples in Fig.15, RoboScribe synthesizes the ordering predicate in Eq.4 for `Tower`. This predicate ensures that any block with a lower goal position must be placed before the current block.

4.3.3 Reinforcement *Program* Learning

Given a program $\mathcal{P}$ inferred from an abstract subtask tree T reflecting the current abstraction of the real robot environment $\mathcal{M}$, the TrainProgram procedure called in Algorithm 5 grounds $\mathcal{P}$ in $\mathcal{M}$ by learning the low-level neural controllers invoked by $\mathcal{P}$ to fulfill the subtasks within T. We maintain separate buffers B_φ to store trajectories associated with each low-level controller π_φ within $\mathcal{P}$. Program trajectories ζ are sampled by executing $\mathcal{P}$ in the real environment $\zeta \sim \texttt{Exec}(\mathcal{M}, \mathcal{P})$. Each sub-trajectory of ζ generated by a specific controller π_φ is stored only in the corresponding buffer B_φ. During each gradient update step, TrainProgram updates each policy π_φ by sampling from its buffer B_φ and optimizes it using any off-the-shelf off-policy RL algorithm (e.g. Soft Actor-Critic), aiming to maximize the expected reward for π_φ:

$$\pi_\varphi = \arg\max_{\pi_\varphi} \mathbb{E}_{\zeta = s_0, a_0, s_1, \cdots, s_L, a_L \sim B_\varphi} \left[\sum_{i=0}^{L} \gamma^t R_\varphi(s_i, a_i) \right]$$

where R_φ denotes the reward function used to train π_φ, L is the sampled trajectory length, and γ is the discount factor.

For each subtask to learn π_φ, our training procedure aims to construct the reward function R_φ that provides feedback based on the satisfaction of these predicates throughout a policy trajectory. Instead of only using φ to provide a binary signal indicating whether a subgoal state has been achieved, R_φ quantifies a continuous measure of state proximity between the current state s and the satisfaction of φ to enables smoother policy optimization. This approach allows for a more granular assessment of progress, guiding the agent incrementally towards the subgoal states in φ. Formally, we define the reward function R_φ recursively based on the structure of the predicate φ:

$$R_\varphi(s) = \begin{cases} R(\varphi_1 \wedge \varphi_2) = \min(R_{\varphi_1}(s), R_{\varphi_2}(s)) & \text{if } \varphi = \varphi_1 \wedge \varphi_2, \\ R(\varphi_1 \vee \varphi_2) = \max(R_{\varphi_1}(s), R_{\varphi_2}(s)) & \text{if } \varphi = \varphi_1 \vee \varphi_2, \\ R(\alpha > \phi) = \alpha(s) - \phi & \text{if } \varphi = \alpha > \phi, \\ R(\alpha < \phi) = \phi - \alpha(s) & \text{if } \varphi = \alpha < \phi. \end{cases}$$

4.4 Extension: Conditional Statements

The GenProgram procedure in Algorithm 5 operates under the assumption that demonstrations implicitly suggest the order for handling multiple objects in a multi-object task. However, this order may vary depending on the goal conditions. For instance, consider a scenario with a peg (blue) and a cube (red) in Fig. 16. If the peg's goal region is above the cube's goal, the task must be completed by first moving the cube, then the peg—and vice versa. Our implementation relaxes this assumption by repurposing the PredicateSynthesis algorithm from Algorithm 6 (designed for sorting objects subsumed by a circular abstract subtask tree node) to synthesize the order under which the multiple predecessors of a tree node for objects of different types should be executed. This approach identifies the conditions for objects to be handled, embedding these conditions into conditional statements to select the appropriate handling sequence.

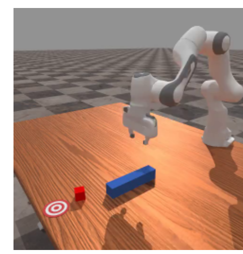


Fig. 16. Conditional Pick&Place Environment for a cube and a peg.

5 Experiments

RoboScribe is implemented in Python. In the implementation, low-level neural policies are Multi-layer Perception (MLP) containing two hidden layers with 256 neurons. We leverage Soft Actor-Critic (SAC) [22] from Stable-Baseline3 [47] as the RL algorithm to train the policies.

Our experiments are designed to answer the following research questions:

- **(RQ1)** Is RoboScribe able to learn effective and interpretable programs?
- **(RQ2)** Does the iterative program learned by RoboScribe generalize to unseen environments without further training?

Main Baselines. Throughout the evaluation, we consider the following baselines:

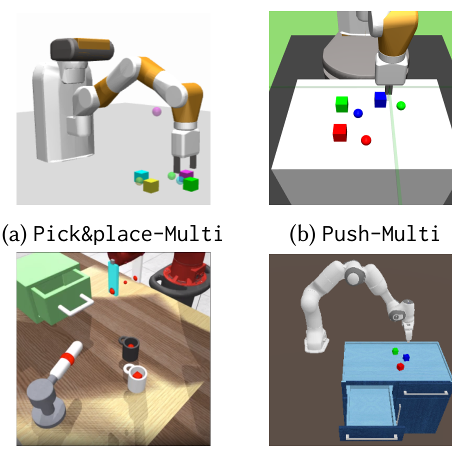
- BC: Behavior Cloning (BC) is a standard learning from demonstration baseline. It applies supervised learning to train a policy that replicates expert actions for given states in demonstrations.
- GAIL [23]: Generative Adversarial Imitation Learning (GAIL) works by alternating between training two components: a discriminator and an agent. The discriminator learns to tell the difference between actions taken by an expert and those taken by the agent in similar situations. Meanwhile, the agent is trained to take actions that make it harder for the discriminator to distinguish between them, encouraging the agent to mimic the expert's behavior. We choose GAIL as a baseline because its discriminator functions similarly to the state abstraction predicates we learn as classifiers.
- goalGAIL [18]: GoalGAIL combines GAIL with Hindsight Experience Replay (HER). Instead of only using the original goal of a trajectory, the goal is replaced with a state that was actually visited during that trajectory. By doing so, the agent receives feedback (or rewards) more frequently.

We select it as a baseline because it is a stronger variant of GAIL that accelerates learning and significantly improves sample efficiency.
- DeepSet [65]: DeepSet embraces an entity-based compositional structure in its neural policy representation based on Self-Attention [59] to leverage the symmetries and invariances in the EFMDP. Like our RoboScribe programs, its policy architectures decompose goal-conditioned tasks into their constituent entities and subgoals.

For fair comparisons, we use the DeepSet architecture for the policies in BC and for both neural networks in GAIL and goalGAIL, improving upon the original MLP-based work. DeepSet's ability to handle an arbitrary number of input objects makes it well-suited for multi-object environments.

We ***exclude direct quantitative comparisons with existing programmatic RL methods*** like PROLEX [42], Tabula [44], and ReGuS [12] because they rely on predefined DSLs with manually crafted state and action abstractions, whereas RoboScribe autonomously discovers these abstractions. This fundamental difference makes direct performance comparisons impractical.

Benchmarks. We use a suite of challenging robot manipulation environments including `Pick&place` (Fig. 1a); `Tower-5` (Fig. 1c) where the goal is to assemble 5 scattered blocks into a tower (88 state dimensions); `Pick&Place-Cond` shown in Fig. 16 where the robot stacks a cube and a peg based on their goal position ordering; `Pick&Place-4` shown in Fig. 17a in the Pick&place-Multi environment where the goal is placing 4 blocks in their designated goal regions on a surface, with the final block needing to be hung by the gripper in the air at its goal position; `Push-3` shown in Fig. 17b in the Push-Multi environment where the goal is pushing 3 blocks to their related goal regions on a table surface; `Meta-World` where a robot needs to be controlled to complete 3 tasks, including pushing the mug back, opening the drawer and turning the faucet left; and `PlaceCubesDrawer` visualized in Fig. 17d. In the challenging `PlaceCubesDrawer` environment (134 state dimensions) from [37], the agent needs to open a drawer and iteratively places three cubes into the drawer. We consider a sparse reward setting in which the agent receives reward 1.0 when the entire task is completed successfully and 0 otherwise. For each environment, we supply a demonstration dataset with 50 successful trajectories. The demonstrations are collected by manually controlling the end effector in a simulator to operate the objects.



(a) `Pick&place-Multi` (b) `Push-Multi`

(c) `Meta-World` (d) `PlaceCubesDrawer`

Fig. 17. Testing environments with multiple entities.

5.1 RQ1: Learning Efficiency and Interpretability

For each environment, we train RoboScribe and the baseline methods with 5 random seeds, reporting their evaluation success rates during training, as shown in Fig. 18. While RoboScribe initially experiences a flat zero success rate early in training, it focuses on comparative abstraction refinement to discover the abstract task structure and grounding abstract actions to reach automatically discovered subgoal conditions, guiding the agent towards the overall goal progressively. RoboScribe's success rate increases rapidly once the program structure is fully developed, eventually surpassing the performance of the baseline methods. For `Tower-5`, there is a sharp increase in

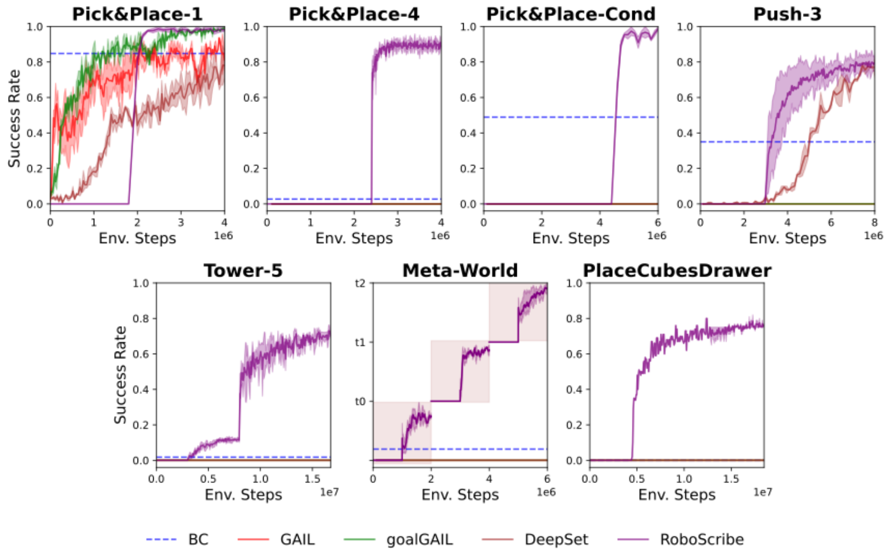


Fig. 18. Rewards for all the tools throughout the training phase. The solid curve represents the mean across 5 random seeds. The shaded area indicates the standard deviation. In Meta-World, we report the success rates for each subtask—pushing the mug back (t0), opening the drawer (t1), and turning the faucet left (t2)—in the order that RoboScribe discovers them.

success rates around 1e7 steps. This is because the task requires the end effector to move its hand away from the top block to a certain height to ensure stable tower construction. The final subgoal of moving away the end effector is relatively easier to learn, and by this point, the agent has already mastered stacking the blocks. As a result, the final task success rate improves significantly after this. Other than `Pick&place` and `Push-3`, the baselines struggle to achieve progress due to the complexity of the observations involving multiple objects and the sparsity of the reward signals.

Compared to black-box neural network policies, the programmatic approach from RoboScribe offers greater interpretability. Programs contain explicit subgoal conditions, making their decision-making process easier to understand. For instance, in the `Push-3` task, where the end effector μ must push a block b to a target position $g(b)$ on a table, RoboScribe synthesizes the state abstraction predicate $\texttt{arctan2}(\mu\downarrow_{x,y} - b\downarrow_{x,y}) - \texttt{arctan2}(g(b)\downarrow_{x,y} - b\downarrow_{x,y}) < \phi$. This predicate captures the condition where the end effector, block, and goal region are aligned for direct pushing, thus providing a clear interpretation of the logic learned by the control strategy. The program also uses state-conditioned loops to define recurring interactions with multiple objects, leading to structured policy representations.

5.2 RQ2: Generalization to New Environments

For the Tower and Push-Multi environments, the capability of handling arbitrary numbers of objects is desired. We analyze the transferability of the synthesized iterative program to diverse environment settings.

Tower Environment. We synthesize the Tower program in a single tower setting with 4 or 5 blocks and evaluate its performance across diverse environments without further training, as shown in Fig. 19. These environments include a taller single tower with 6 or 7 blocks, multiple towers with 2 to 3 blocks per tower, and a pyramid tower with 4 to 9 blocks. As the baselines

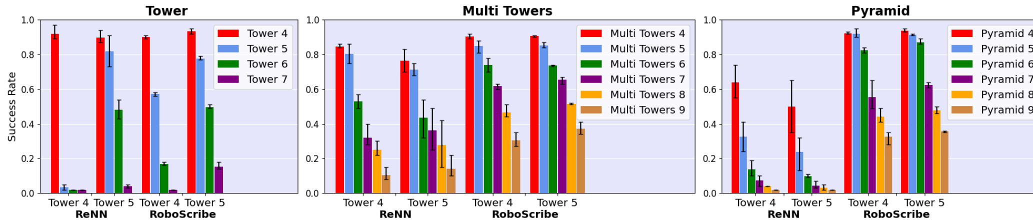
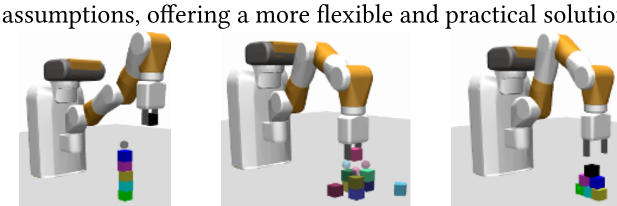


Fig. 20. Comparison between ReNN and RoboScribe on zero-shot generalization to new Tower environment settings. Specifically, policies trained on single tower with 4 blocks or 5 blocks are evaluated on Single (but taller) towers, multiple towers and pyramid configurations with varying numbers of blocks. Success rate is reported as accuracy of completing a task averaged over 500 episodes.

discussed in Sec. 5.1 fail to solve the Tower task, we turn to the curriculum learning-based approach ReNN [30], which progressively learns to stack 2, 3, and ultimately 5 blocks, while RoboScribe directly learns to handle all 5 blocks in one go. Unlike ReNN, which requires expert-designed curriculum, RoboScribe operates without such assumptions, offering a more flexible and practical solution. Fig. 20 presents the results for RoboScribe and ReNN [30]. In the single tower setting, ReNN benefits from curriculum learning, achieving slightly better results when the training and evaluation environments match (e.g., single towers with 4 and 5 blocks). However, RoboScribe demonstrates superior generalization. For instance, when transferring a policy trained on 4 blocks to a taller single tower, ReNN achieves less than 5% success, while RoboScribe trained on 4 blocks achieves 55% (±1%) success with 5 blocks and 15% (±1%) with 6 blocks. In both multi-tower and pyramid settings, RoboScribe significantly outperforms ReNN across block counts from 4 to 9.



(a) `Single Tower` (b) `Multi Towers` (c) `Pyramids`

Fig. 19. Novel Tower Environments.

Push-Multi Environment. To evaluate the iterative program learned in the Push-Multi environment, we introduce a confined version, `Push-Multi Confined` (Fig. 21), where goals are randomly arranged along a line near the table's upper edge. To achieve the goal condition, the robot benefits from pushing the blocks in a certain order. For example, in Fig. 21, the robot should push the blue block first, followed by green, then red. Pushing blocks out of this order, such as green or red first, may obstruct the blue block's path to its goal. For RoboScribe, we reuse the learned iterative program by providing demonstrations of the correct entity handling sequence, allowing it to update its loop condition without additional training. In contrast, we continue training the baseline DeepSet [65] model on the confined environment until convergence, as it cannot structurally update its model like RoboScribe. RoboScribe correctly learns the entity handling order from the demonstration and updates the loop condition accordingly. In comparison, DeepSet achieves a success rate of 81.3% (±4.0%), while RoboScribe achieves 86.3% (±0.5%), averaged over 500 episodes. RoboScribe's superior performance demonstrates the generalization of learned policies and program structures. Additionally, with the

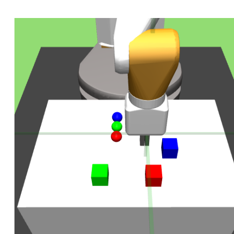


Fig. 21. `Push-Multi Confined` Environment.

interpretability of its programmatic policy, RoboScribe offers greater flexibility in transferring the synthesized program to different environment settings.

6 Related Work

Programmatic Reinforcement Learning. Our work is closely related to recent advance on exploring domain-specific programs as an interpretable representation for RL. PIRL [60, 61] and Viper [5] synthesize loop-free, stateless programs, which face limitations in complex robot tasks. Inala et al. [26] improved on this by learning robot controllers as state machines, enabling generalization to tasks with repeating behaviors. These methods rely heavily on strong supervision from oracles like pretrained RL controllers. The tasks they can solve are thus bounded by the capability of the oracle. In contrast, program synthesis methods such as PROLEX [42] and Tabula [44] learn robot control programs from task demonstrations. They generalize these demonstrations into regex-based sketches or Mealy automata to bootstrap synthesis. They can synthesize programs with control flow structures including loops and conditionals, allowing generalization from a specific sequence of actions to a general structure to solve unseen tasks. Their DSLs feature extensive library functions for manipulating various objects and teleporting robots to different locations. LEAPS [58], PRL [45], and ReGuS [12] eliminate the need for pretrained oracles and synthesizes robot-control programs directly from reward signals. They demonstrate that utilizing rich control-flow constructs (state-conditioned loops and procedure calls) can effectively tackle long-horizon and sparse-reward tasks, which are beyond the capabilities of standard deep RL baselines. However, these existing works rely on a manually designed library of state abstraction predicates and abstract actions to bootstrap synthesis. RoboScribe addresses the primary challenge of automating the construction of robot state and action abstractions.

Learning State and Action Abstraction. RoboScribe shares similarities with generalized planning methods like [54, 57], which derive looped plans for solving unbounded problem instances. However, planning techniques require a provided state transition model for each robot action within the abstract state space. Component-based synthesis techniques with user-defined predicates, as in [10, 20], have similar requirements. Automatically learning state and action abstractions has been explored in task and motion planning for robot control [6, 21, 27, 41]. Existing techniques typically learn either predicates from demonstrations assuming low-level controllers are given [13, 34, 53] or learn controllers from demonstrations assuming known predicates [1, 15, 52]. RoboScribe simultaneously learns state and action abstractions, removing such assumptions. There exist library learning techniques [8, 9, 19] that use *syntax abstraction* to extract common structures from a program corpus as reusable library functions. In contrast, RoboScribe performs *state abstraction*. RoboScribe is broadly related to hierarchical RL and planning for robot learning [3, 28, 29, 38, 40, 43, 46, 54, 62]. However, such techniques often struggle with long-horizon tasks with sparse rewards. Traditional controller synthesis algorithms, especially those using formal methods and temporal logic, rely on automata-based approaches involving abstraction and discretization of continuous state and action spaces [14, 49]. These methods face limitations in high-dimensional systems, where discretization can lead to issues like state explosion.

Reward-guided Program Synthesis. Synthesis algorithms often design dense rewards to guide search directions. For example, Probe [4] and Syntia [7] evaluate programs using input-output examples, generating rich rewards based on output similarity. Faery [11] employs Monte Carlo estimation to sample user queries for additional examples. However, in sparse-reward scenarios, synthesizing complete programs with complex control flow through Monte Carlo methods is challenging due to the low probability of discovering programs with nonzero rewards. RoboScribe addresses this by using comparative abstraction refinement to learn state abstraction predicates that capture subgoal conditions, effectively breaking down the learning process.

Conclusion. This paper introduces RoboScribe, a program synthesis framework guided by abstraction refinement to address long-horizon, multi-object tasks in robotics. RoboScribe alternates between comparative abstraction refinement and iterative program learning, using demonstrations and execution trajectories from synthesized programs to iteratively refine environment abstractions until a task-solving program can be generated. It identifies recurring subroutines from raw, continuous state-action spaces without predefined abstractions. Experimental results show that RoboScribe generalizes effectively to long-horizon tasks with varying object counts, outperforming baseline methods in interpretability and efficiency. Currently, our language restricts predicates to use norm and arctan functions. It remains an open question whether this expressiveness is sufficient for all tasks—for example, whether additional trigonometric functions are needed, which we leave for future work.

Data-Availability Statement

The intended artifact for this work is an implementation of RoboScribe, a program synthesis framework for robotic control tasks. The artifact includes both the codebase for synthesizing iterative programs and associated datasets used for training and evaluation. The artifact, along with relevant datasets, will be made available upon acceptance for Artifact Evaluation. A prelimnary version is available at https://anonymous.4open.science/r/AbsDemo-D008/.

References

[1] Diego Aineto, Sergio Jiménez, and Eva Onaindia. 2022. A Comprehensive Framework for Learning Declarative Action Models. *J. Artif. Intell. Res.* 74 (2022), 1091–1123. https://doi.org/10.1613/JAIR.1.13073

[2] David Andre and Stuart J. Russell. 2002. State Abstraction for Programmable Reinforcement Learning Agents. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence.*

[3] Pierre-Luc Bacon, Jean Harb, and Doina Precup. 2017. The Option-Critic Architecture. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*

[4] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.* OOPSLA (2020).

[5] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *Advances in Neural Information Processing Systems, NeurIPS 2018.*

[6] Pascal Bercher, Ron Alford, and Daniel Höller. 2019. A Survey on Hierarchical Planning - One Abstract Idea, Many Concrete Realizations. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 6267–6275. https://doi.org/10.24963/IJCAI.2019/875

[7] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium, USENIX Security 2017.*

[8] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. 2023. Top-Down Synthesis for Library Learning. *Proc. ACM Program. Lang.* 7, POPL (2023), 1182–1213. https://doi.org/10.1145/3571234

[9] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. babble: Learning Better Abstractions with E-Graphs and Anti-unification. *Proc. ACM Program. Lang.* 7, POPL (2023), 396–424. https://doi.org/10.1145/3571207

[10] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. 2020. Program Synthesis Using Deduction-Guided Reinforcement Learning. In *Computer Aided Verification - 32nd International Conference, CAV 2020.*

[11] Yanju Chen, Chenglong Wang, Xinyu Wang, Osbert Bastani, and Yu Feng. 2023. Fast and Reliable Program Synthesis via User Interaction. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023.*

[12] Guofeng Cui, Yuning Wang, Wenjie Qiu, and He Zhu. 2024. Reward-Guided Synthesis of Intelligent Agents with Control Structures. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1730–1754. https://doi.org/10.1145/3656447

[13] Aidan Curtis, Tom Silver, Joshua B. Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. 2022. Discovering State and Action Abstractions for Generalized Task and Motion Planning. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022.* AAAI Press, 5377–5384. https://doi.org/10.1609/AAAI.V36I5.20475

[14] Mehdi Dadvar, Rashmeet Kaur Nayyar, and Siddharth Srivastava. 2023. Conditional Abstraction Trees for Sample-Efficient Reinforcement Learning. In *The 39th Conference on Uncertainty in Artificial Intelligence*. https://openreview.net/forum?id=tQP094M0j8G

[15] Joaquim Ortiz de Haro, Jung-Su Ha, Danny Driess, and Marc Toussaint. 2021. Structured deep generative models for sampling on constraint manifolds in sequential manipulation. In *Conference on Robot Learning, 8-11 November 2021, London, UK (Proceedings of Machine Learning Research, Vol. 164)*, Aleksandra Faust, David Hsu, and Gerhard Neumann (Eds.). PMLR, 213–223. https://proceedings.mlr.press/v164/ortiz-haro22a.html

[16] Quentin Delfosse, Wolfgang Stammer, Thomas Rothenbacher, Dwarak Vittal, and Kristian Kersting. 2023. Boosting Object Representation Learning via Motion and Object Continuity. In *Machine Learning and Knowledge Discovery in Databases: Research Track - European Conference, ECML PKDD 2023*.

[17] Thomas G. Dietterich. 2000. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *J. Artif. Intell. Res.* (2000).

[18] Yiming Ding, Carlos Florensa, Pieter Abbeel, and Mariano Phielipp. 2019. Goal-conditioned Imitation Learning. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 15298–15309. https://proceedings.neurips.cc/paper/2019/hash/c8d3a760ebab631565f8509d84b3b3f1-Abstract.html

[19] Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Lucas Morales, Luke B. Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. [n. d.]. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021*. 835–850.

[20] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*.

[21] Caelan Reed Garrett, Rohan Chitnis, Rachel M. Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. 2021. Integrated Task and Motion Planning. *Annu. Rev. Control. Robotics Auton. Syst.* 4 (2021), 265–293. https://doi.org/10.1146/ANNUREV-CONTROL-091420-084139

[22] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*. PMLR, 1861–1870.

[23] Jonathan Ho and Stefano Ermon. 2016. Generative Adversarial Imitation Learning. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 4565–4573. https://proceedings.neurips.cc/paper/2016/hash/cc7e2b878868cbae992d1fb743995d8f-Abstract.html

[24] Jarrett Holtz, Arjun Guha, and Joydeep Biswas. 2020. Robot Action Selection Learning via Layered Dimension Informed Program Synthesis. In *4th Conference on Robot Learning, CoRL 2020, 16-18 November 2020, Virtual Event / Cambridge, MA, USA (Proceedings of Machine Learning Research, Vol. 155)*, Jens Kober, Fabio Ramos, and Claire J. Tomlin (Eds.). PMLR, 1471–1480. https://proceedings.mlr.press/v155/holtz21a.html

[25] Baichuan Huang, Abdeslam Boularias, and Jingjin Yu. 2022. Parallel Monte Carlo Tree Search with Batched Rigid-body Simulations for Speeding up Long-Horizon Episodic Robot Planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2022, Kyoto, Japan, October 23-27, 2022*. IEEE, 1153–1160. https://doi.org/10.1109/IROS47612.2022.9981962

[26] Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. 2020. Synthesizing Programmatic Policies that Inductively Generalize. In *8th International Conference on Learning Representations, ICLR 2020*.

[27] George Dimitri Konidaris, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. 2018. From Skills to Symbols: Learning Symbolic Representations for Abstract High-Level Planning. *J. Artif. Intell. Res.* 61 (2018), 215–289. https://doi.org/10.1613/JAIR.5575

[28] Youngwoon Lee, Shao-Hua Sun, Sriram Somasundaram, Edward S. Hu, and Joseph J. Lim. 2019. Composing Complex Skills by Learning Transition Policies. In *7th International Conference on Learning Representations, ICLR 2019*.

[29] Youngwoon Lee, Jingyun Yang, and Joseph J. Lim. 2020. Learning to Coordinate Manipulation Skills via Skill Behavior Diversification. In *8th International Conference on Learning Representations, ICLR 2020*.

[30] Richard Li, Allan Jabri, Trevor Darrell, and Pulkit Agrawal. 2020. Towards Practical Multi-Object Manipulation using Relational Reinforcement Learning. In *2020 IEEE International Conference on Robotics and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020*. IEEE, 4051–4058. https://doi.org/10.1109/ICRA40945.2020.9197468

[31] Junchi Liang and Abdeslam Boularias. 2023. Learning Category-Level Manipulation Tasks from Point Clouds with Dynamic Graph CNNs. In *Proceedings of the 2023 International Conference on Robotics and Automation (ICRA)*.

[32] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as Policies: Language Model Programs for Embodied Control. In *IEEE International Conference on Robotics and*

Automation, ICRA 2023, London, UK, May 29 - June 2, 2023. IEEE, 9493–9500. https://doi.org/10.1109/ICRA48891.2023.10160591

[33] Zhixuan Lin, Yi-Fu Wu, Skand Vishwanath Peri, Weihao Sun, Gautam Singh, Fei Deng, Jindong Jiang, and Sungjin Ahn. 2020. SPACE: Unsupervised Object-Oriented Scene Representation via Spatial Attention and Decomposition. In *8th International Conference on Learning Representations, ICLR 2020*.

[34] João Loula, Kelsey R. Allen, Tom Silver, and Josh Tenenbaum. 2020. Learning constraint-based planning models from demonstrations. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021*. IEEE, 5410–5416. https://doi.org/10.1109/IROS45743.2020.9341535

[35] Chaitanya Mitash, Kostas E. Bekris, and Abdeslam Boularias. 2017. A self-supervised learning system for object detection using physics simulation and multi-view pose estimation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2017*.

[36] Chaitanya Mitash, Abdeslam Boularias, and Kostas E. Bekris. 2018. Robust 6D Object Pose Estimation with Stochastic Congruent Sets. In *British Machine Vision Conference 2018, BMVC 2018*.

[37] Tongzhou Mu, Zhan Ling, Fanbo Xiang, Derek Yang, Xuanlin Li, Stone Tao, Zhiao Huang, Zhiwei Jia, and Hao Su. 2021. ManiSkill: Generalizable Manipulation Skill Benchmark with Large-Scale Demonstrations. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/eda80a3d5b344bc40f3bc04f65b7a357-Abstract-round2.html

[38] Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. 2018. Data-Efficient Hierarchical Reinforcement Learning. In *Annual Conference on Neural Information Processing Systems, NeurIPS 2018*.

[39] Ashvin Nair, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. 2017. Overcoming Exploration in Reinforcement Learning with Demonstrations. *CoRR* abs/1709.10089 (2017). arXiv:1709.10089 http://arxiv.org/abs/1709.10089

[40] Soroush Nasiriany, Vitchyr Pong, Steven Lin, and Sergey Levine. 2019. Planning with Goal-Conditioned Policies. In *Annual Conference on Neural Information Processing Systems, NeurIPS 2019*.

[41] Hanna M. Pasula, Luke S. Zettlemoyer, and Leslie Pack Kaelbling. 2007. Learning Symbolic Models of Stochastic Domains. *J. Artif. Intell. Res.* 29 (2007), 309–352. https://doi.org/10.1613/JAIR.2113

[42] Noah Patton, Kia Rahmani, Meghana Missula, Joydeep Biswas, and Isil Dillig. 2024. Programming-by-Demonstration for Long-Horizon Robot Tasks. *Proc. ACM Program. Lang.* 8, POPL (2024), 512–545. https://doi.org/10.1145/3632860

[43] Xue Bin Peng, Michael Chang, Grace Zhang, Pieter Abbeel, and Sergey Levine. 2019. MCP: Learning Composable Hierarchical Control with Multiplicative Compositional Policies. In *Annual Conference on Neural Information Processing Systems, NeurIPS 2019*.

[44] David Porfirio, Laura Stegner, Maya Cakmak, Allison Sauppé, Aws Albarghouthi, and Bilge Mutlu. 2023. Sketching Robot Programs On the Fly. In *Proceedings of the 2023 ACM/IEEE International Conference on Human-Robot Interaction, HRI 2023, Stockholm, Sweden, March 13-16, 2023*, Ginevra Castellano, Laurel D. Riek, Maya Cakmak, and Iolanda Leite (Eds.). ACM, 584–593. https://doi.org/10.1145/3568162.3576991

[45] Wenjie Qiu and He Zhu. 2022. Programmatic Reinforcement Learning without Oracles. In *10th International Conference on Learning Representations, ICLR 2022*.

[46] Ahmed Hussain Qureshi, Jacob J. Johnson, Yuzhe Qin, Taylor Henderson, Byron Boots, and Michael C. Yip. 2020. Composing Task-Agnostic Policies with Deep Reinforcement Learning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*.

[47] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. 2021. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research* 22, 268 (2021), 1–8. http://jmlr.org/papers/v22/20-1364.html

[48] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016*. 779–788.

[49] Wei Ren, Raphaël M. Jungers, and Dimos V. Dimarogonas. 2024. Zonotope-Based Symbolic Controller Synthesis for Linear Temporal Logic Specifications. *IEEE Trans. Autom. Control.* 69, 11 (2024), 7630–7645. https://doi.org/10.1109/TAC.2024.3394313

[50] Rahul Shome, Wei N. Tang, Changkyu Song, Chaitanya Mitash, Hristiyan Kourtev, Jingjin Yu, Abdeslam Boularias, and Kostas E. Bekris. 2019. Towards Robust Product Packing with a Minimalistic End-Effector. In *International Conference on Robotics and Automation, ICRA 2019, Montreal, QC, Canada, May 20-24, 2019*. IEEE, 9007–9013. https://doi.org/10.1109/ICRA.2019.8793966

[51] Tom Silver, Kelsey R. Allen, Alex K. Lew, Leslie Pack Kaelbling, and Josh Tenenbaum. 2020. Few-Shot Bayesian Imitation Learning with Logical Program Policies. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*.

[52] Tom Silver, Ashay Athalye, Joshua B. Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. 2022. Learning Neuro-Symbolic Skills for Bilevel Planning. In *Conference on Robot Learning, CoRL 2022, 14-18 December 2022, Auckland, New Zealand (Proceedings of Machine Learning Research, Vol. 205)*, Karen Liu, Dana Kulic, and Jeffrey Ichnowski (Eds.). PMLR, 701–714. https://proceedings.mlr.press/v205/silver23a.html

[53] Tom Silver, Rohan Chitnis, Nishanth Kumar, Willie McClinton, Tomás Lozano-Pérez, Leslie Pack Kaelbling, and Joshua B. Tenenbaum. 2023. Predicate Invention for Bilevel Planning. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, Brian Williams, Yiling Chen, and Jennifer Neville (Eds.). AAAI Press, 12120–12129. https://doi.org/10.1609/AAAI.V37I10.26429

[54] Tom Silver, Rohan Chitnis, Joshua B. Tenenbaum, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. 2021. Learning Symbolic Operators for Task and Motion Planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2021*.

[55] Avishai Sintov, Andrew S. Morgan, Andrew Kimmel, Aaron M. Dollar, Kostas E. Bekris, and Abdeslam Boularias. 2019. Learning a State Transition Model of an Underactuated Adaptive Hand. *IEEE Robotics Autom. Lett.* 4, 2 (2019), 1287–1294. https://doi.org/10.1109/LRA.2019.2894875

[56] Changkyu Song and Abdeslam Boularias. 2019. Object Rearrangement with Nested Nonprehensile Manipulation Actions. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2019, Macau, SAR, China, November 3-8, 2019*. IEEE, 6578–6585. https://doi.org/10.1109/IROS40897.2019.8967548

[57] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. 2011. A new representation and associated algorithms for generalized planning. *Artif. Intell.* (2011).

[58] Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. 2021. Learning to Synthesize Programs as Interpretable and Generalizable Policies. In *Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021*.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[60] Abhinav Verma, Hoang Minh Le, Yisong Yue, and Swarat Chaudhuri. 2019. Imitation-Projected Programmatic Reinforcement Learning. In *Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*.

[61] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. 2018. Programmatically Interpretable Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*.

[62] Alexander Vezhnevets, Volodymyr Mnih, Simon Osindero, Alex Graves, Oriol Vinyals, John P. Agapiou, and Koray Kavukcuoglu. 2016. Strategic Attentive Writer for Learning Macro-Actions. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*.

[63] Jimmy Xin, Linus Zheng, Kia Rahmani, Jiayi Wei, Jarrett Holtz, Isil Dillig, and Joydeep Biswas. 2024. Programmatic Imitation Learning From Unlabeled and Noisy Demonstrations. *IEEE Robotics Autom. Lett.* 9, 6 (2024), 4894–4901. https://doi.org/10.1109/LRA.2024.3385691

[64] Yichen Yang, Jeevana Priya Inala, Osbert Bastani, Yewen Pu, Armando Solar-Lezama, and Martin Rinard. 2021. Program Synthesis Guided Reinforcement Learning for Partially Observed Environments. In *Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021*.

[65] Allan Zhou, Vikash Kumar, Chelsea Finn, and Aravind Rajeswaran. 2024. Policy Architectures for Compositional Generalization in Control. In *Proceedings of the 1st Reinforcement Learning Conference, RLC 2024, Amherst, MA, USA, August 9-12, 2024*, Philip S. Thomas, Feryal M. P. Behbahani, Glen Berseth, Scott M. Jordan, Scott Niekum, Andrew Patterson, Eugene Vinitsky, Adam White, Martha White, and Amy Zhang (Eds.). University of Massachusetts Amherst, MA, USA, 2264–2283.

[66] Shaojun Zhu, Andrew Kimmel, Kostas E. Bekris, and Abdeslam Boularias. 2018. Fast Model Identification via Physics Engines for Data-Efficient Policy Search. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, Jérôme Lang (Ed.). ijcai.org, 3249–3256. https://doi.org/10.24963/ijcai.2018/451