Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2026.DOI

Macro-Operator Generation and Predicate Selection for TAMP Operator Learning

CAN EMIR BORA[1] AND EMRE UGUR[1] (Senior Member, IEEE)

[1]Department of Computer Engineering, Bogazici University, Istanbul 34342, Turkey (e-mail: can.bora@std.bogazici.edu.tr, emre.ugur@bogazici.edu.tr)

Corresponding author: Can Emir Bora (e-mail: can.bora@std.bogazici.edu.tr).

This work was supported by the INVERSE project (101136067) funded by the European Union and by the STAR program funded by TUBITAK.

ABSTRACT Creating symbolic operators by hand is one of the main bottlenecks in deploying Task and Motion Planning systems (TAMP). Recent works show that these operators can instead be learned directly from demonstration data. Existing methods, however, typically learn each action in isolation and cannot capture the recurring multi-step structure of manipulation tasks, so the search becomes intractable on long sequential tasks. A further inefficiency arises in the symbolic state: every provided predicate is evaluated at every search node, even when it never appears in any learned operator. We present a system that addresses both problems together. Its central component is the automatic generation of macro-operators, composite actions that compress a recurring sequence of individual actions into a single planning step. Our system discovers causally linked action pairs directly from the training data, where one action produces exactly the condition that the next one requires, and turns each pair into a new operator. Alongside this, our system prunes every predicate that no learned operator references, which shrinks the symbolic state evaluated at each search node. Together, these changes shorten the effective planning horizon, and the benefit they bring grows with the length of the task. Across four TAMP domains, our method reaches up to a $\sim 4.6\times$ planning speedup compared to the baseline method, namely Learning Operators for TAMP. More importantly, it solves a long sequential task that the baseline cannot solve. Macro-operator discovery thus not only accelerates planning but, in certain domains, determines solvability in practice.

INDEX TERMS Bilevel Planning, Macro-Operator Discovery, Manipulation Planning, Operator Learning, Symbolic Planning, Task and Motion Planning

I. INTRODUCTION

AUTONOMOUS robots in real-world environments face complex planning tasks. High-level reasoning involves deciding which objects to interact with and in what sequence, while low-level tasks require precise geometric decisions, such as determining grasp poses, motion trajectories, and placement locations. This tight coupling between discrete choices and continuous parameters makes robotic planning fundamentally challenging [1].

Task and Motion Planning (TAMP) [2] addresses this challenge through a bilevel approach: symbolic planning operators define an abstract transition model that guides search over action sequences, while a low-level search refines these abstract plans by assigning continuous values (e.g., grasp poses, placement locations) to each step. By using classical AI planning methods, TAMP can leverage heuristics to guide the search and quickly discard infeasible plans, exploring far fewer options than brute-force enumeration [3].

A significant constraint of TAMP systems is their dependence on manually crafted symbolic operators, which necessitate considerable domain knowledge for accurate specification [4]. Learning can ease this burden, and it has been applied throughout the TAMP pipeline, from samplers that propose grasp poses and placement locations [5]–[7] to guidance models that steer the symbolic search toward promising action sequences [8], [9]. These methods make the search more efficient, but the operators themselves are still written by hand, so the core specification burden remains.

To address exactly this problem, Learning Operators for Task and Motion Planning (LOFT) [10] demonstrates that the operators themselves can be learned directly from transition experience, continuing a long line of action-model learning in classical planning [11]–[13]. The learned operators follow the style of the Planning Domain Definition Language

(PDDL) [14], a standard formalism that describes each action through its preconditions and effects, defined over symbolic facts called *predicates*. They are also probabilistic, since each possible effect is assigned a probability of occurring. At planning time, these operators guide a symbolic search that proposes *plan skeletons*, action sequences whose continuous parameters, such as grasp poses and placement locations, are left open and filled in afterwards by a sampling procedure.

Although this shows that operators can be learned effectively for TAMP, the method comes with several assumptions and limitations. The most significant one is that it treats every individual action in isolation and does not sufficiently address the fact that some actions almost always occur together in a fixed causal order, with one action producing exactly the condition that enables the next. Because of this, the planner must rediscover these recurring two-step patterns from scratch at every search node, which inflates the effective planning horizon and causes the branching factor to explode on long-horizon tasks. A second limitation is that every provided predicate is evaluated at every search node even when it never appears in any learned operator, which wastes state parsing effort and enlarges the symbolic search space.

In this work, we address these limitations one by one with the following contributions:

- **Generating macro-operators:** To address the isolated treatment of individual actions, we automatically generate *macro-operators* from the training data, that is, single composite actions that each replace a short and fixed sequence of individual actions. For this, we identify pairs of actions that are causally linked, where the first produces a condition that the second consumes, and register each pair as a new composite operator. This collapses frequent two-step sub-routines into one planning step. As a result, the effective planning horizon shrinks and the branching factor grows more slowly on long-horizon tasks.
- **Pruning unused predicates:** To address the fact that every provided predicate is evaluated at every search node even when many of them never appear in any learned operator, we propose *Iterative Predicate Selection (IPS)*, a post-learning step that automatically removes predicates absent from all learned preconditions and effects. This shrinks the symbolic state the planner must parse and reason over at every node.

We evaluate our method on four TAMP domains. Three of them, **Cover**, **Blocks**, and **Painting**, have relatively short plan lengths and are also used in the baseline work [10]. In addition, in order to verify the method on long-horizon tasks, we introduce **Kitchen**, a cooking domain that we designed following the kitchen tasks used in prior TAMP work [6], [15], in which seven ingredients must each be picked, chopped, cooked, and plated (Figure 1). Kitchen demands long sequences of interdependent actions, since every ingredient has to pass through several processing stations in a fixed order. Experiments show that our contributions achieve up to $\sim 4.6\times$ planning speedup and enable solving the Kitchen domain. The baseline cannot solve this domain, and no plan is found there even when the planning time limit is removed. A component-wise ablation study further quantifies the contribution of each proposed component.

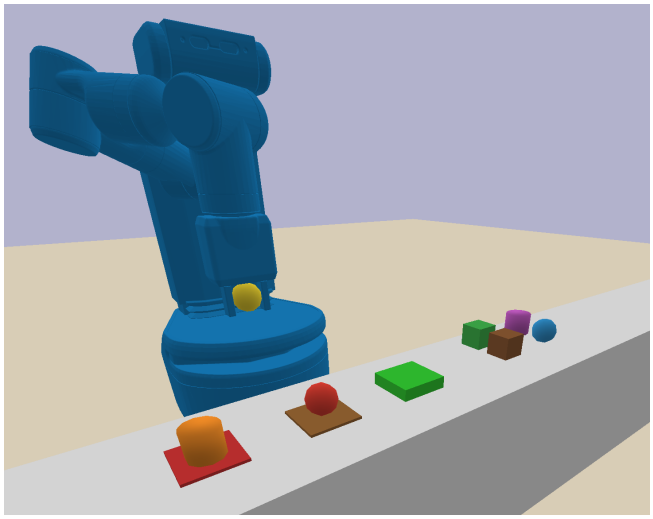

FIGURE 1. The 7-ingredient Kitchen domain in PyBullet. A Fetch robot manipulates seven ingredients, represented by distinct colored shapes, across three workstations: Burner (red pad), Chopping Board (brown pad), and Dish (green pad). Four ingredients wait at the far end of the counter while three are actively being processed. Each ingredient must be picked, chopped, cooked, and plated to satisfy the goal.

II. RELATED WORK

A. TASK AND MOTION PLANNING (TAMP)

Following the taxonomy of Garrett et al. [1], approaches to TAMP can be broadly categorized based on how symbolic reasoning interacts with continuous variables during the planning process. One family formulates TAMP as a single large-scale optimization problem. It optimizes over hybrid trajectories, that is, trajectories that combine discrete choices with continuous motion, and it integrates logic and geometry directly [16], [17]. A second family avoids this joint formulation and instead relies on sampling-based procedures that follow a search-then-sample pattern. These methods first search for a symbolic plan and then sample continuous values to realize it [15], [18], [19]. In these methods, symbolic operators, often represented in PDDL [14], are used to generate a *plan skeleton*, a sequence of high-level actions with their discrete arguments fixed but continuous parameters left open. A low-level continuous optimizer or sampler then attempts to fill in those continuous arguments. A nearly universal characteristic of these deeply integrated TAMP systems is their reliance on manually authored symbolic operators, whose preconditions and effects must be specified by a human expert [4]. Our work builds on the premise that these symbolic abstractions can instead be learned automatically from demonstration and interaction data.

B. LEARNING FOR TAMP AND OPERATOR DISCOVERY

To reduce the burden of manual specification, learning has been applied to several parts of the TAMP pipeline. At the geometric level, one line of work learns continuous samplers, i.e. generative models that propose promising grasp poses or placement locations from the current state, so that the low-level optimizer explores fewer dead ends [5]–[7]. At the symbolic level, a complementary line learns search guidance. Kim and Shimanuki [8] learn a value function over relational states that scores symbolic actions and steers the tree search. Driess et al. [9] train a sequence model that predicts whole high-level action sequences from an image of the scene. These approaches learn *how to search* but still rely on hand-written operators describing what each action does.

Closer to our setting is the model-based paradigm of learning the operators themselves, i.e. their preconditions and effects. This has a long history in classical planning [11]. Examples include noisy deictic rules, a probabilistic rule format that tolerates noisy and uncertain action effects [12]; LOCM (Learning Object-Centred Models), which extracts action models from plan traces [20]; STRIPS-style learning, which recovers action models in the classic STRIPS (Stanford Research Institute Problem Solver) representation with the help of classical planners [13]; and sparse relational transition models learned from data [21]. Whereas these methods assume the symbolic predicates are given, a more recent thread discovers the predicates themselves from robot interaction. Ahmetoglu et al. introduce *DeepSym*, which trains a deep encoder-decoder to predict action effects [22]. This network turns continuous observations into discrete object symbols, and PDDL operators are then extracted over those symbols. Follow-up work extends the idea to multi-object scenes using attention layers that expose object roles and relations, producing relational predicates that generalize across object counts [23], [24]. Similarly, other efforts learn abstractions tailored to planning, each starting from a different representation: symbols grounded in an agent's skills [25], entity-centric state abstractions for model-based reinforcement learning [26], and neuro-symbolic predicates optimized end-to-end for planning performance [27]. Most relevant to our setting, Huang et al. [28] infer a planning domain, that is, the preconditions and effects of each action, from a small number of demonstrations given at test time. A recent survey places these efforts in the broader context of combining learning and planning [29]. LOFT [10] belongs to this operator-learning family but targets TAMP directly: it learns lifted, object-parameterized operators from a handful of demonstrations and uses them inside a sampling-based TAMP planner. However, it suffers from the two limitations set out in Section I: each action is learned in isolation, and every provided predicate is evaluated at every search node. We address both in this paper. Since it keeps the operator structure explicit, it is also a natural foundation for the components we propose, and we adopt it as our baseline.

C. MACRO-OPERATORS IN PLANNING

Macro-operators are composite actions formed by chaining several individual actions into a single planning step. They have a long history in classical AI planning: early systems such as MACROPS, a mechanism in the STRIPS planner that stored successful action sequences for later reuse, showed that this can substantially reduce the depth of future searches [30], [31]. Subsequent work proposed various criteria for selecting useful macros, for example by analyzing which action pairs recur across problem instances or which state transitions are repeated in training data [32], [33].

In robotics, the same idea appears under the name of macro-actions, which group repetitive low-level behaviors into reusable skills and simplify control [34], [35]. Applied to TAMP, macros can collapse multi-step sub-routines such as picking up an object, moving it, and placing it into a single operator, which shortens the effective planning horizon. However, for *learned* probabilistic TAMP operators, the automatic discovery and validation of such macros remain underexplored: existing frameworks learn each action in isolation and ignore the sequential structure present in the demonstration data [10], [12], [13], [21]. We address this gap by discovering causally linked action pairs directly from training trajectories and registering each pair as a new operator, which the operator learner then treats identically to individual actions during precondition learning and probability estimation.

III. METHOD

Figure 2 shows the complete system we propose, from the training data to an executable plan. Our main contributions are shown with the stages colored with orange boxes, each of which is presented in its own subsection below. Before detailing our contributions, we first give an overview of the background methods used in our pipeline, namely Operator Learning and TAMP Planning, in the next subsection.

A. BACKGROUND

In this section, we provide the background methods on which our work is built. In our TAMP approach, at the high level, a symbolic planner searches with operators written in PDDL. Each operator has preconditions and effects defined over *predicates*, the symbolic facts that abstract the continuous state into a discrete representation. Using A* search [36] with the heuristic [37] that estimates goal distance by summing the costs of achieving each subgoal independently, this stage returns candidate *plan skeletons*: action sequences whose discrete arguments are fixed but whose continuous parameters are left open. At the low level, a backtracking search turns a skeleton into an executable plan. For each step, it calls domain-specific samplers to propose continuous values, then simulates the action to check feasibility, and reverts to an earlier step whenever the current one fails. Each step is given a fixed number of sampling attempts before it is considered a failure. If no assignment works for a skeleton, the high-level planner produces the next plan.

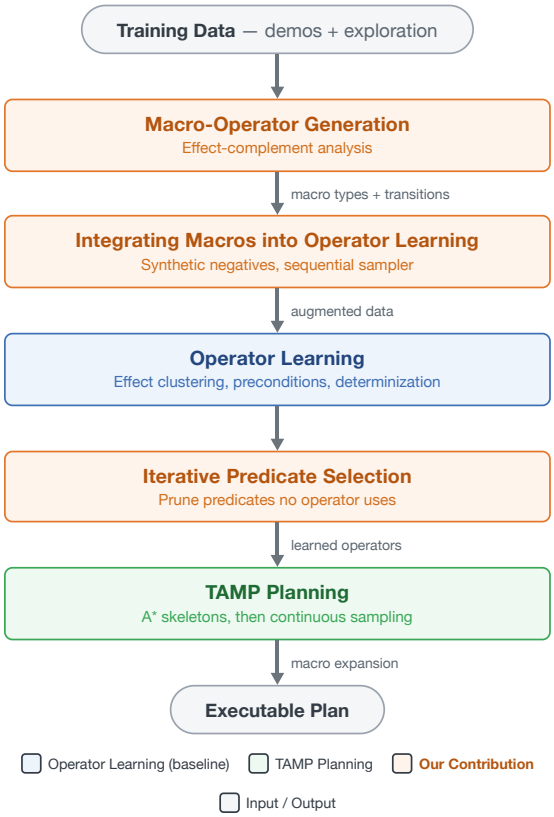

FIGURE 2. Architecture of the full pipeline, from training data to an executable plan.

Next, we describe the *operator learning module*, for which we use LOFT [10]. The input is a dataset of low-level transitions, together with a fixed set of user-provided typed predicates. The transitions come from two sources. The oracle demonstrations $\mathcal{D}$ are successful task solutions, whereas the random-exploration set $\mathcal{R}$ is collected by taking random actions from states visited in the demonstrations, thereby providing the negative examples used during learning. The module first converts every continuous state into a symbolic state using a deterministic function, Parse, that returns the subset of predicates that hold in that state. Learning then proceeds in three stages. *(1) Lifted effect clustering* computes the predicates that the action adds and the predicates it deletes. Lifting, here, corresponds to replacing the object instances with typed variables. Transitions with the same effects but different object instances are grouped into one cluster, and each cluster becomes a candidate operator. *(2) Precondition learning* then finds, for each cluster, the preconditions under which its effect occurs. For this, it scores the sets of observed predicates before each action. A candidate set is scored positively when it is consistently observed in the cluster's own transitions, and negatively when it is observed in transitions for the same action that produced different effects. Note that these negative transitions come mostly from the random exploration data. At the end, the best-scoring set becomes the operator's precondition. *(3) Parameter estimation* computes each effect's probability by counting how often the effect occurs and dividing by the number of times the action was applied: $\hat{p}_k = n_k/N$. Effects whose estimated probability falls below a threshold are discarded. The remaining outcomes are converted into deterministic operators via all-outcome determinization [38], yielding one deterministic operator per retained outcome.

B. OUR METHOD

1) Macro-Operator Generation

We generate macro-operators automatically through what we call *effect-complement analysis*. The goal is to find pairs of actions that may be causally linked. Two actions form such a pair when the first consistently adds a predicate that the second consistently deletes. The first action thus creates exactly the condition that the second one needs, and then removes it. We detect these pairs directly from the training trajectories as explained in Algorithm 1. In the rest of this subsection, we will detail this procedure step by step, with references to this algorithm.

Terminology. Before describing the procedure, we distinguish two terms used throughout. An *action type* is a template such as Pick or Stack, whose arguments are not yet bound to concrete objects. An *action* is one grounded instance of such a template, such as Pick(b, r), in which the arguments refer to specific objects. The first step is to characterize what each action type consistently changes in the symbolic state.

Phase 1: Consistent effects of each action type. The aim of this phase (lines 1-8) is to find consistent effects of actions from the oracle demonstration set $\mathcal{D}$, as the non-consistent and rare effects are caused by sampling variance rather than by the action itself. We deliberately exclude the random-exploration set $\mathcal{R}$ at this stage, since most of its actions fail and change nothing, which would dilute these ratios and hide effects that the action does produce reliably. $\mathcal{D}$ includes transitions in the form of (x_i, a_i, x_{i+1}): a continuous state x_i, the action a_i applied in it, and the resulting continuous state x_{i+1}. For every transition, we compute the symbolic states $s_i = \text{Parse}(x_i)$ and $s_{i+1} = \text{Parse}(x_{i+1})$, where Parse is the function introduced in Section III-A. A predicate is *added* if it is absent in s_i but present in s_{i+1}, that is $s_{i+1} \setminus s_i$. It is *deleted* if it is present in s_i but absent in s_{i+1}, that is $s_i \setminus s_{i+1}$. Grouping transitions by their action type, we count how often each predicate is added or deleted. A predicate is considered to be a *consistent add* of action type a if it is added in more than half of a's transitions, and a *consistent delete* if it is deleted in more than half of the times. These two sets, denoted by $\text{Adds}[a]$ and $\text{Dels}[a]$, correspond to the output of Phase 1.

Phase 2: Causal pairs. The aim of this phase (lines 9-15) is to find action types that are causally linked, where one action produces a condition that the next one needs. A good macro should join exactly such a pair, since merging them removes an intermediate step the planner would otherwise have to rediscover. In order to find the set of predicates that one action consistently creates and another consistently removes, we

Algorithm 1 Macro-Operator Generation

Input: $\mathcal{D}$: oracle transitions (x_i, a_i, x_{i+1}); $\mathcal{R}$: random-exploration transitions (x_i, a_i, x_{i+1}); $\mathcal{A}$: action types
Output: $\mathcal{A}_M$: discovered macro action types; $\mathcal{T}_+$: macro transitions

// Phase 1: Finding consistent effects of each action type
1: **for** each transition $(x_i, a_i, x_{i+1}) \in \mathcal{D}$ **do**
2: $s_i \leftarrow \text{Parse}(x_i)$; $s_{i+1} \leftarrow \text{Parse}(x_{i+1})$ ▷ Parse returns the predicates that hold in a continuous state
3: record added predicates $s_{i+1} \setminus s_i$ and deleted predicates $s_i \setminus s_{i+1}$ for action type a_i
4: **end for**
5: **for** each action type $a \in \mathcal{A}$ **do**
6: $\text{Adds}[a] \leftarrow \{\, p \mid \Pr[p \text{ added} \mid a] > 0.5 \,\}$
7: $\text{Dels}[a] \leftarrow \{\, p \mid \Pr[p \text{ deleted} \mid a] > 0.5 \,\}$
8: **end for**
// Phase 2: Causal pairs
9: $\mathcal{C} \leftarrow \emptyset$
10: **for** each ordered pair of action types (a_i, a_{i+1}) with $a_i \neq a_{i+1}$ **do**
11: $\mathcal{L}(a_i, a_{i+1}) \leftarrow \text{Adds}[a_i] \cap \text{Dels}[a_{i+1}]$ ▷ predicates a_i creates and a_{i+1} removes
12: **if** $\mathcal{L}(a_i, a_{i+1})$ contains a predicate of arity > 0 **then**
13: $\mathcal{C} \leftarrow \mathcal{C} \cup \{(a_i, a_{i+1})\}$
14: **end if**
15: **end for**
// Phase 3: Validation and conflict resolution
16: $f(a_i, a_{i+1}) \leftarrow$ number of times $(a_i \rightarrow a_{i+1})$ occurs consecutively in $\mathcal{D} \cup \mathcal{R}$
17: $\mathcal{V} \leftarrow \{\, (a_i, a_{i+1}) \in \mathcal{C} \mid f(a_i, a_{i+1}) > 0 \,\}$
18: **for** each pair with both (a_i, a_{i+1}) and (a_{i+1}, a_i) in $\mathcal{V}$ **do**
19: remove from $\mathcal{V}$ the direction with the smaller f
20: **end for**
// Phase 4: Build macro operators
21: $\mathcal{A}_M \leftarrow \emptyset$; $\mathcal{T}_+ \leftarrow \emptyset$
22: **for** each consecutive $(x_i, a_i, x_{i+1}, a_{i+1}, x_{i+2})$ in $\mathcal{D} \cup \mathcal{R}$ with $(a_i, a_{i+1}) \in \mathcal{V}$ **do**
23: Create macro action M by unifying the arguments of a_i and a_{i+1}
24: $\mathcal{A}_M \leftarrow \mathcal{A}_M \cup \{M\}$; $\mathcal{T}_+ \leftarrow \mathcal{T}_+ \cup \{(x_i, M, x_{i+2})\}$
25: **end for**
26: **return** $\mathcal{A}_M$, $\mathcal{T}_+$

consider every ordered pair of action types (a_i, a_{i+1}), where the subscripts denote the order in which the two would be applied rather than positions in a particular trajectory. For each such pair, we compute the causal-link set $\mathcal{L}(a_i, a_{i+1}) = \text{Adds}[a_i] \cap \text{Dels}[a_{i+1}]$. A non-empty $\mathcal{L}(a_i, a_{i+1})$ means that a_i produces a condition that a_{i+1} then consumes, which is exactly the causal link we aim to find. In case a non-empty $\mathcal{L}(a_i, a_{i+1})$ contains only zero-arity predicates, i.e., predicates without object arguments, they are discarded because such global state predicates carry no object-specific causal link, and keeping them would create spurious pairings between unrelated actions. The output of this phase is the causal pair set $\mathcal{C}$ that contains action pairs with produced-and-consumed predicates with at least one object argument (e.g. Holding(block) or On(block, table)).

Phase 3: Validation and conflict resolution. The aim of this phase (lines 16-20) is to identify the causal pairs with empirical support, as some pairs in $\mathcal{C}$ might be plausible on paper, but may never actually occur in the data. In order to validate that they are in the data, we count how often each candidate $(a_i, a_{i+1}) \in \mathcal{C}$ appears as two consecutive actions in the data. We use both the oracle demonstrations $\mathcal{D}$ and the random exploration set $\mathcal{R}$ for this. Unlike Phase 1, this step counts occurrences rather than ratios, so the additional sequences in $\mathcal{R}$ can only add evidence.

Let $f(a_i, a_{i+1})$ denote this count. We store only the candidates that occur at least once, giving the validated set $\mathcal{V} = \{(a_i, a_{i+1}) \in \mathcal{C} \mid f(a_i, a_{i+1}) > 0\}$. After verification, we resolve the conflicting cases when both (a_i, a_{i+1}) and (a_{i+1}, a_i) appear in $\mathcal{V}$. Registering both would create two macros for the same underlying pattern, thereby inflating the search branching factor. Whenever both directions are present, we keep only the one with the higher frequency f and discard the other pair.

Phase 4: Building macro operators. The goal of this phase (lines 21-26) is to turn each verified pair into one usable macro operator. This requires two components: a single argument list for the macro, and training transitions from which the next stage can learn. Both are obtained by scanning the data for occurrences of the pair. The first difficulty is that the two sub-actions have separate argument lists. Some objects appear in both actions, others in only one. To behave as a single operator, the macro needs one merged list in which a shared object occupies a single slot. We call this *argument unification*. Consider a Pick followed by a Stack in the Blocks domain: $a_i = \text{Pick}(b_1)$ picks up block b_1, and $a_{i+1} = \text{Stack}(b_1, b_2)$ places it onto block b_2. Here b_1 is shared, and b_2 is specific to Stack, so the merged argument list is (b_1, b_2). To recall how this list maps back to each sub-action, we store two index vectors: $\mathbf{i}_1 = (1)$ for Pick, which uses only b_1, and $\mathbf{i}_2 = (1, 2)$ for Stack, which uses both. These vectors let the planner recover the original arguments of each sub-action when the macro is later split back into its two steps at execution time.

The macro operator M produced this way is added to the macro action set $\mathcal{A}_M$. We then build the training data for M, that is, the transitions from which its preconditions and effects will later be learned. For every consecutive occurrence $(x_i, a_i, x_{i+1}, a_{i+1}, x_{i+2})$ in $\mathcal{D} \cup \mathcal{R}$ with $(a_i, a_{i+1}) \in \mathcal{V}$, we record a positive transition (x_i, M, x_{i+2}). This transition skips the intermediate state x_{i+1}, so the macro appears to be a single step that takes the world from x_i directly to x_{i+2}. These transitions form the set $\mathcal{T}_+$.

Note that the set of shared objects can differ from one occurrence to another, so the same pair can yield more than one macro, each with its own argument pattern and its own preconditions and effects learned in the next stages.

Generation leaves us with the macro action types $\mathcal{A}_M$ and their positive transitions $\mathcal{T}_+$. At this point a macro is only a typed argument list together with a set of transitions, and not yet a usable operator. It has no learned preconditions or effects, no procedure for proposing continuous parameters for its two sub-actions, and no means of execution on the robot,

since it does not belong to the original action set. The next section describes how we integrate the generated macros into the operator learning module to obtain all three.

2) Integrating Macros into the Operator Learning Module

Negative Example Generation. To learn accurate preconditions, the module requires both positive and negative examples: states where the macro's effect occurred, and states where it did not [12]. Phase 4 already provides the positives $\mathcal{T}_+$, but no negatives exist for macros. For individual actions, such negatives are already available: the random exploration set $\mathcal{R}$ contains many states in which an action was attempted and produced no effect. Macros, however, were not part of the action set when $\mathcal{R}$ was collected, so $\mathcal{R}$ holds no such record for them. We must therefore generate macro negatives ourselves, and we do so by manufacturing states in which a macro clearly does not apply.

To this end, for each macro type $M \in \mathcal{A}_M$ we draw a random state x from $\mathcal{R}$ and bind the macro's parameters to objects $\mathbf{o}$ of matching types. We then record the transition $(x, M(\mathbf{o}), x)$, in which the state is left unchanged. A macro that genuinely applied would alter the state, so an unchanged state is direct evidence that the macro's preconditions did not hold in x. This makes $(x, M(\mathbf{o}), x)$ a valid negative, and these transitions form the negative set $\mathcal{T}_-$.

Learning Macro Operators. To learn the generated macros without modifying the module, we present each one in the form it already expects for an ordinary action. It takes three inputs: action types $\mathcal{A}$, demonstration transitions $\mathcal{D}$, and random-exploration transitions $\mathcal{R}$ that supply negatives. We augment each with its macro counterpart: $\mathcal{A}' = \mathcal{A} \cup \mathcal{A}_M$ (generated macro types), $\mathcal{D}' = \mathcal{D} \cup \mathcal{T}_+$ (positive transitions from Phase 4), and $\mathcal{R}' = \mathcal{R} \cup \mathcal{T}_-$ (synthetic negatives). Each macro is therefore treated as an ordinary action type, and the module learns its preconditions and effects through the same three stages it applies to individual actions. The only macro-specific addition is a *sequential sampler*, described next.

Sequential Continuous Sampling. A macro must still propose continuous values, such as a grasp pose for its first sub-action and a placement location for its second. These two are not independent: a valid placement depends on where the object was grasped. Sampling them separately would ignore this dependency and produce many infeasible combinations. We therefore give each macro a *sequential sampler* that respects the order of the two sub-actions. It samples the first sub-action's parameters, simulates that sub-action to reach the intermediate state x_{i+1}, and then samples the second sub-action's parameters conditioned on x_{i+1}. The index vectors from Phase 4 tell the sampler which arguments belong to each sub-action, and any parameter shared by both is sampled once and reused. The same index vectors are used once more after planning, when every macro in the returned plan is expanded back into its two sub-actions to give an executable plan over the original action set $\mathcal{A}$.

3) Iterative Predicate Selection (IPS)

The two preceding subsections together form our first contribution. Our second contribution targets the symbolic state itself. As described in the Background, the module turns each continuous state into a symbolic one by evaluating every predicate in the user-provided set $\mathcal{P}$ through Parse. The planner does this at every node it expands, so the cost of $\mathcal{P}$ is paid repetitively throughout the search. The set is also fixed before learning starts, which means the user must choose it without knowing which predicates the learned operators will end up using.

These two facts together create the problem. Once learning has finished, some predicates in $\mathcal{P}$ turn out to appear in no learned precondition and in no learned effect. Such a predicate is still evaluated at every node, and it still enlarges the grounded state, that is, the symbolic state instantiated with all concrete objects in the scene. Yet no operator ever reads it or changes it, so the facts it produces are ones the planner can do nothing with.

IPS removes them once operator learning is complete. Drawing on the principle of feature selection in machine learning [39], we retain only the predicates that occur in at least one learned operator. Writing $\mathcal{O}$ for the learned operator set and $\mathrm{Preds}[o]$ for the predicates appearing in the preconditions or effects of an operator o, the retained set is

$$\mathcal{P}' \leftarrow \mathcal{P} \cap \bigcup_{o \in \mathcal{O}} \mathrm{Preds}[o]. \tag{1}$$

The planner then parses states with $\mathcal{P}'$ in place of $\mathcal{P}$. The operators themselves are left untouched, and the filter costs a single scan over $\mathcal{O}$, which is negligible beside learning.

IV. EXPERIMENTS

A. EXPERIMENTAL SETUP

Our evaluation spans four TAMP domains, from simple tabletop manipulation to complex multi-stage cooking. **Cover** is the simplest: place two blocks on targets, performed in 2–4 steps. **Blocks** raises the difficulty by requiring six blocks to be stacked into goal configurations in approximately 10 steps. **Painting** adds object attributes such as color and dryness, producing 8 operators and plans of approximately 32 steps. **Kitchen** is the hardest: seven ingredients, each requiring pick, chop, cook, and plate in strict sequence, yielding plan lengths that overwhelm search when only individual actions are available.

Operator learning is performed with the implementation of the baseline [10], and PyBullet [40] serves as the physics backend. Training data consists of expert demonstrations, where a planner with full access to the ground-truth operators solves each task, together with random exploration trajectories that expose the agent to a broader set of state transitions. The resulting probabilistic operators learned from this data are then determinized as described in Section III-A for use with A* search. Each domain uses 20 test problems evaluated over 5 independent seeds, with timeouts ranging from 1 to 10 s per problem.

We compare the original baseline against our complete pipeline, referred to as **Ours**, which integrates both contributions presented in Section III: macro-operator generation (Sections III-B1 and III-B2) and Iterative Predicate Selection (Section III-B3).

B. SYSTEM BEHAVIOR: LEARNED OPERATORS AND MACRO DISCOVERY

We examine macro discovery in detail in the Blocks domain, whose action set is small enough that every discovered macro can be checked by hand against what a person would expect.

Effect-complement analysis. We apply the effect-complement analysis of Section III-B1 to this domain. Table 1 shows the outcome: five causal candidates emerge, and resolving reverse-ordered pairs by frequency leaves two validated macros, Pick→PutOnTable (34.1% of consecutive pairs) and Pick→Stack (28.0%). Both rest on the same causal link: Pick adds Holding(b), which PutOnTable and Stack each delete as their first precondition. This outcome is the desired one. In this domain, a picked block can only be placed on the table or stacked on another block, and these are exactly the two macros the analysis retains. Without any manual guidance, the procedure thus discovers all meaningful pick-and-place routines of the domain and discards only the spurious reverse orderings.

TABLE 1. Effect-complement macro discovery in the Blocks domain. Five causal candidates are identified; symmetric-pair resolution retains the two highest-frequency directions.

Candidate Pair	Shared Predicate(s)	Frequency	Retained
Pick → PutOnTable	`Holding`	34.1%	✓
Pick → Stack	`Clear, Holding`	28.0%	✓
PutOnTable → Pick	`Clear`	21.9%	(symmetric)
PutOnTable → Stack	`Clear`	9.8%	(symmetric)
Stack → Pick	`Clear, On`	6.1%	(symmetric)

Unified macro operators. Here we examine what the module actually learns for a validated pair. For each pair, the arguments of the two sub-actions are first unified into a single list (Section III-B1), and the module then learns the macro's preconditions and effects from the augmented dataset, with no manual guidance. As an example, consider the Pick→Stack macro for the case where block b_1 starts On another block b_3. Here Clear(b) means nothing is stacked on block b, On(b, b') means b rests on b', and HandEmpty means the gripper holds nothing. The learned operator takes the following form:

`MacroPickStack(`b_1`:block, `b_2`:block, `b_3`:block)`
`PRE: Clear(`b_1`) ∧ Clear(`b_2`) ∧ HandEmpty`
`∧ On(`b_1`,`b_3`)`
`ADD: On(`b_1`,`b_2`) ∧ Clear(`b_3`)`
`DEL: Clear(`b_2`) ∧ On(`b_1`,`b_3`) ∧ HandEmpty`

Read together, the operator says: starting from b_1 on b_3 with both b_1 and b_2 clear and the gripper empty, the macro ends with b_1 on b_2 and b_3 now clear. This is exactly what a domain engineer would write by hand for "pick b_1 off b_3 and stack it on b_2", with the intermediate Holding state absorbed inside the macro. A second variant, for the case where b_1 starts OnTable rather than on another block, is discovered automatically. The precondition search treats the two situations as distinct patterns without any manual guidance.

Planning implications. Adding these macros brings the total action set from 4 individual operators to 7. Executed plan lengths stay roughly the same (10.0 ± 1.1 steps after macro expansion versus 10.2 ± 1.1 for the baseline), but the A* search is considerably faster, dropping from 0.185 ± 0.912 s to 0.040 ± 0.059 s. Because each macro collapses two decisions into one, the effective branching factor at every step is reduced, and the planner reaches goal-relevant states with fewer search nodes. A representative solved plan illustrates this:

`[MacroPickPutOnTable(`b_5`, pose),`
`MacroPickPutOnTable(`b_4`, pose),`
`MacroPickStack(`b_4`, `b_3`), MacroPickStack(`b_5`, `b_4`), ...]`

In the Kitchen domain, the same discovery process validates 3 causal pairs, namely Pick followed by each of the three placement actions (PlaceOnBoard, PlaceOnBurner, PlaceOnDish). Because these pairs occur with two different argument-sharing patterns, they yield 6 macro operators and 16 in total (10 individual + 6 macro). Without these macros, each of the 7-ingredient pick-chop-cook-plate workflows requires more than 50 individual steps, and the expanded plans our pipeline returns average 55.6 steps. This plan length overwhelms the A* search budget entirely. With macro operators, however, the same goals are achieved in 0.051 ± 0.019 s across all 20 test problems. To confirm that the baseline failure is not merely a timeout issue, we ran it with the timeout disabled; no plan was found after 30 minutes on a single Kitchen problem.

C. BASELINE VS. FULL PIPELINE

In this section we compare our full pipeline against the baseline on all four domains, in terms of success rate, planning time, and the structure of the learned operator sets (Tables 2 and 3).

Success rates. Both approaches achieve $100.0 \pm 0.0\%$ on Painting. In Blocks, the baseline achieves $99.0 \pm 2.0\%$. The single failure across five seeds occurs when the backtracking sampler exhausts its budget on a step whose geometric constraints (e.g. a tight grasp pose) are unusually hard to satisfy. Our full pipeline achieves a perfect $100.0 \pm 0.0\%$ on this domain, because macros shorten the plan skeleton and the sampler therefore encounters fewer steps at which it can fail. In Cover the ordering reverses, and our pipeline drops to $98.0 \pm 4.0\%$; we return to this domain below. The most consequential difference arises in Kitchen: every approach without macro operators achieves 0.0% success, while our macro-augmented pipeline solves all 100 test instances (20 problems × 5 seeds) without a single failure. These numbers show that macros do not cost reliability. On the three short domains the success rate stays within two points of the baseline. On Kitchen the gap is not small but total, since the baseline solves nothing there.

TABLE 2. Planning performance of the baseline versus our full pipeline. Plan time and plan length are mean ± std over 5 seeds (20 problems each). "Failed" indicates 0.0% success or no available measurement.

	Plan Time (s)		Plan Length (steps)	
Domain	Baseline	Ours	Baseline	Ours
Cover	$\mathbf{0.001 \pm 0.001}$	0.093 ± 0.190	2.6 ± 0.0	2.6 ± 0.1
Blocks	0.185 ± 0.912	$\mathbf{0.040 \pm 0.059}$	10.2 ± 1.1	10.0 ± 1.1
Painting	0.028 ± 0.026	$\mathbf{0.024 \pm 0.007}$	31.8 ± 0.4	31.8 ± 0.4
Kitchen	Failed	$\mathbf{0.051 \pm 0.019}$	Failed	55.6 ± 0.0

Planning time. The impact of macro operators is most visible in Blocks, where our method achieves 0.040 ± 0.059 s versus the baseline's 0.185 ± 0.912 s, a $\sim 4.6\times$ speedup arising from the reduced number of A* nodes that must be expanded to reach the goal. In Kitchen, our method plans in 0.051 ± 0.019 s while the baseline fails entirely. Painting shows a small gain (0.028 ± 0.026 s versus 0.024 ± 0.007 s), as it is solved near-instantly regardless of the approach. Cover is the only domain where the cost of macros exceeds their benefit: our method needs 0.093 ± 0.190 s against the baseline's 0.001 ± 0.001 s. Plans there are only two to four steps long, so the planner already finishes in about a millisecond, and the discovered Pick→Place macro adds a further branch at every search node without any depth left to remove. The success rate is affected as well, although the effect is small in absolute terms: two of the 100 instances are left unsolved, which is within the variation we observe across seeds. This is the known trade-off in macro planning [32]. A macro is worthwhile only when the depth it removes outweighs the branching it adds. In a domain this small there is almost no depth left to remove. Across the four domains the pattern is clear. The benefit of macros grows with plan length: a net loss on two-step tasks, a speedup on ten-step tasks, and the difference between failure and success on the longest one.

Structural properties. Table 3 shows operator counts, predicate counts after IPS, and training times. Macro generation adds 3 operators in Blocks (one Pick→PutOnTable and two Pick→Stack variants), 1 in Cover, 1 in Painting, and 6 in Kitchen. The number of macro operators exceeds the number of validated pairs whenever a pair occurs with different argument-sharing patterns, as explained in Section III-B1. IPS (Section III-B3) reduces the active predicate count in Cover ($5 \rightarrow 3$) and Blocks ($6 \rightarrow 5$) by removing predicates unreferenced by any learned operator. This directly shrinks the grounded state representation computed at each A* node. In Painting and Kitchen, all provided predicates appear in the learned rules, so IPS has no effect. Training times scale with operator set complexity: Painting requires ~ 3.8 s due to its 8-operator, 14-predicate state space, while all other domains train in under 0.5 s. Macro generation adds modest overhead (e.g., $0.069 \rightarrow 0.196$ s in Blocks) for generating macro transitions and learning preconditions for the additional operators. The cost of the method is therefore small. At most six operators are added, and training stays below four seconds in every domain. Both costs are paid once, before planning starts. The benefit returns at every node of every later search.

TABLE 3. Structural properties of learned operator sets: operator count, predicate count after IPS, and training time. Operator learning succeeds in every domain, including Kitchen, where it is planning rather than learning that fails without macros.

	Operators		Predicates		Train Time (s)	
Domain	Baseline	Ours	Baseline	Ours	Baseline	Ours
Cover	4	5	5	**3**	0.013	0.021
Blocks	4	7	6	**5**	0.069	0.196
Painting	8	9	14	14	3.829	3.918
Kitchen	10	16	8	8	0.271	0.424

D. ABLATION STUDY: ISOLATING COMPONENT CONTRIBUTIONS

We test a *subtractive* ablation: start from the full model and remove one component at a time. This isolates the contribution of each component. We run the ablation on Blocks, where both components contribute but the effect is modest, and on Kitchen, where the macro component is the deciding factor.

Blocks domain. Removing macros has the largest effect, as Table 4 shows. The success rate decreases from 100.0 to $99.0 \pm 2.0\%$ and planning time grows $4.4\times$, from 0.041 to 0.180 s. Removing IPS leaves the success rate at 100.0% but adds one predicate back to the state, which carries a small cost at every A* node and raises planning time from 0.041 to 0.042 s. The two components work at different levels. IPS removes a fixed cost from every node. Macros reduce how many nodes are visited.

Kitchen domain. Table 5 shows that Kitchen separates the two components more clearly. Removing IPS leaves performance essentially unchanged, since every provided predicate already appears in some learned operator and there is nothing left to prune. Removing the macros, however, takes the success rate to $0.0 \pm 0.0\%$ across every seed and problem, with no plan found even when the timeout is lifted (Section IV-B). The two ablations together explain what macro operators provide.

TABLE 4. Subtractive ablation on Blocks (5 seeds). Removing the macro component is the only modification that degrades both success rate and planning time.

Variant	Success (%)	Plan Time (s)	Operators	Predicates
Ours	$\mathbf{100.0 \pm 0.0}$	$\mathbf{0.041 \pm 0.060}$	7	5
Ours w/o IPS	100.0 ± 0.0	0.042 ± 0.060	7	6
Ours w/o Macro	99.0 ± 2.0	0.180 ± 0.886	4	5

TABLE 5. Subtractive ablation on Kitchen (5 seeds). Only removing macro operators causes complete failure.

Variant	Success (%)	Plan Time (s)	Operators
Ours	$\mathbf{100.0 \pm 0.0}$	$\mathbf{0.050 \pm 0.018}$	16
Ours w/o IPS	100.0 ± 0.0	0.051 ± 0.021	16
Ours w/o Macro	$\mathbf{0.0 \pm 0.0}$	Failed	10

Their gain is not a fixed percentage. It grows with the length of the task, and beyond a certain horizon it is the only reason a plan is found.

V. CONCLUSION

Learning symbolic operators from data removes one of the main bottlenecks in deploying TAMP systems, yet existing methods leave two practical barriers in place: a symbolic state cluttered with predicates that no learned operator ever uses, and a planning horizon that search with individual actions alone cannot overcome. This work closes both gaps. We proposed an automated macro-operator generation pipeline that discovers causally linked action pairs through effect-complement analysis. We also proposed Iterative Predicate Selection, which prunes up to 40% of the unreferenced predicates and cuts the parsing overhead paid at every A* node.

Taken together, the experiments suggest that the benefit of macro-operators is not a fixed speedup but a shortened effective planning horizon, whose value scales with the length of the task. In very short, two-step problems, the added branching outweighs that gain, so our method does not bring any advantage of the baseline method. In longer, for example 10-step problems, our method can achieve a $\sim 4.6\times$ speedup over the baseline method. Moreover, in tasks, for example, of more than 50 steps, it determines whether a plan is found at all: baseline individual-action planner can not solve that domain within thirty minutes, while our pipeline solves it in ~ 0.05 s. Therefore, for long-horizon manipulation, finding which actions belong together is therefore a requirement for planning to work rather than a late optimization.

Our current approach leaves several directions open. Our generation pipeline currently considers only pairs of consecutive actions. Extending it to longer chains or nested sequences (e.g., discovering full pick-wash-dry-place routines) would enable deeper plan abstraction. A second direction concerns the predicates themselves: our pipeline prunes unused predicates but cannot create new ones. Discovering new predicates automatically from sensor data would reduce the remaining domain engineering effort. Similarly, the continuous samplers that propose grasp poses and placement locations are currently hand-designed; learning them from interaction data would remove another manual component. All experiments in this work run in simulation using PyBullet [40]. Deploying the learned operators on a real robot would introduce noisy perception, imprecise execution, and sim-to-real transfer challenges that the current pipeline does not address. Evaluating robustness under these conditions is an important next step. Finally, the current system learns offline: it first collects all training data, then mines macros and learns operators in a single batch. An online variant that updates its operator set incrementally as the robot encounters new situations would be more practical for long-lived deployment, where the task distribution may shift over time.

REFERENCES

[1] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Perez, "Integrated task and motion planning," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 4, pp. 265–293, 2021.

[2] S. Cambon, R. Alami, and F. Gravot, "A hybrid approach to intricate motion, manipulation and task planning," *The International Journal of Robotics Research*, vol. 28, no. 1, pp. 104–126, 2009.

[3] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "FFRob: Leveraging symbolic planning for efficient task and motion planning," *The International Journal of Robotics Research*, vol. 37, no. 1, pp. 104–136, 2018.

[4] Y. Zhu, J. Tremblay, S. Birchfield, and Y. Zhu, "Hierarchical planning for long-horizon manipulation with geometric and symbolic scene graphs," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 6541–6548.

[5] B. Kim, L. P. Kaelbling, and T. Lozano-Pérez, "Learning to guide task and motion planning using score-space representation," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 2810–2817.

[6] Z. Wang, C. R. Garrett, L. P. Kaelbling, and T. Lozano-Pérez, "Active model learning and diverse action sampling for task and motion planning," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 4107–4114.

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

[8] B. Kim and L. Shimanuki, "Learning value functions with relational state representations for guiding task-and-motion planning," in *Conference on Robot Learning*. PMLR, 2020, pp. 955–968.

[9] D. Driess, J.-S. Ha, and M. Toussaint, "Deep visual reasoning: Learning to predict action sequences for task and motion planning from an initial scene image," in *Proceedings of Robotics: Science and Systems (RSS)*, 2020.

[10] T. Silver, R. Chitnis, J. Tenenbaum, L. P. Kaelbling, and T. Lozano-Perez, "Learning symbolic operators for task and motion planning," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 3182–3189.

[11] A. Arora, H. Fiorino, D. Pellier, M. Métivier, and S. Pesty, "A review of learning planning action models," *The Knowledge Engineering Review*, vol. 33, 2018.

[12] H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling, "Learning symbolic models of stochastic domains," *Journal of Artificial Intelligence Research*, vol. 29, pp. 309–352, 2007.

[13] D. Aineto, S. Jiménez, and E. Onaindia, "Learning strips action models with classical planning," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 28, 2018, pp. 399–407.

[14] M. Fox and D. Long, "Pddl2.1: An extension to pddl for expressing temporal planning domains," *Journal of Artificial Intelligence Research*, vol. 20, pp. 61–124, 2003.

[15] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 30, 2020, pp. 440–448.

[16] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning." in *IJCAI*, 2015, pp. 1930–1936.

[17] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, "Incremental task and motion planning: A constraint-based approach." in *Robotics: Science and Systems (RSS)*, 2016.

[18] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 1470–1477.

[19] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 639–646.

[20] S. N. Cresswell, T. L. McCluskey, and M. M. West, "Acquiring planning domain models using locm," in *The Knowledge Engineering Review*, vol. 28, no. 2, 2013, pp. 195–213.

[21] V. Xia, Z. Wang, K. R. Allen, T. Silver, and L. P. Kaelbling, "Learning sparse relational transition models," in *International Conference on Learning Representations*, 2019.

[22] A. Ahmetoglu, M. Y. Seker, J. Piater, E. Oztop, and E. Ugur, "Deepsym: Deep symbol generation and rule learning for planning from unsupervised

robot interaction," *Journal of Artificial Intelligence Research*, vol. 75, pp. 709–745, 2022.
[23] A. Ahmetoglu, B. Celik, E. Oztop, and E. Ugur, "Discovering predictive relational object symbols with symbolic attentive layers," *IEEE Robotics and Automation Letters*, vol. 9, no. 2, pp. 1977–1984, 2024.
[24] A. Ahmetoglu, E. Oztop, and E. Ugur, "Symbolic manipulation planning with discovered object and relational predicates," *IEEE Robotics and Automation Letters*, vol. 10, no. 2, pp. 1968–1975, 2025.
[25] G. Konidaris, L. P. Kaelbling, and T. Lozano-Perez, "From skills to symbols: Learning symbolic representations for abstract high-level planning," *Journal of Artificial Intelligence Research*, vol. 61, pp. 215–289, 2018.
[26] R. Veerapaneni, J. D. Co-Reyes, M. Chang, M. Janner, C. Finn, J. Wu, J. B. Tenenbaum, and S. Levine, "Entity abstraction in visual model-based reinforcement learning," in *Conference on Robot Learning (CoRL)*, 2020.
[27] T. Silver, R. Chitnis, N. Kumar, W. McClinton, T. Lozano-Perez, L. P. Kaelbling, and J. B. Tenenbaum, "Predicate invention for bilevel planning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 10, 2023, pp. 12 120–12 129.
[28] J. Huang, A. Tao, R. Marco, M. Bogdanovic, J. Kelly, and F. Shkurti, "Automated planning domain inference for task and motion planning," in *2025 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2025, pp. 12 534–12 540.
[29] E. Ugur, A. Ahmetoglu, Y. Nagai, T. Taniguchi, M. Saveriano, and E. Oztop, "Neuro-symbolic robotics," TechRxiv, jun 2025. [Online]. Available: https://doi.org/10.36227/techrxiv.175000864.40759665/v1
[30] R. E. Fikes, P. E. Hart, and N. J. Nilsson, "Learning and executing generalized robot plans," *Artificial intelligence*, vol. 3, no. 1-3, pp. 251–288, 1972.
[31] R. E. Korf, *Learning to Solve Problems by Searching for Macro-Operators*. Pitman, 1985.
[32] A. Botea, M. Enzenberger, M. Müller, and J. Schaeffer, "Macro-FF: Improving AI planning with automatically learned macro-operators," *Journal of Artificial Intelligence Research*, vol. 24, pp. 581–621, 2005.
[33] M. H. Newton, J. Levine, M. Fox, and D. Long, "Learning macro-actions for arbitrary planners and domains," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2007.
[34] E. Gizzi, M. G. Castro, and J. Sinapov, "Creative problem solving by robots using action primitive discovery," in *International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*. IEEE, 2019, pp. 228–233.
[35] V. Sarathy, D. Kasenberg, S. Goel, J. Sinapov, and M. Scheutz, "Spotter: Extending symbolic planning operators through targeted reinforcement learning," in *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 2021, pp. 1118–1126.
[36] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.
[37] B. Bonet and H. Geffner, "Planning as heuristic search," *Artificial Intelligence*, vol. 129, no. 1-2, pp. 5–33, 2001.
[38] S. W. Yoon, A. Fern, and R. Givan, "Ff-replan: A baseline for probabilistic planning," in *ICAPS*, vol. 7, 2007, pp. 352–359.
[39] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.
[40] E. Coumans and Y. Bai, "PyBullet, a Python module for physics simulation for games, robotics and machine learning," GitHub, 2016.

CAN EMIR BORA is currently pursuing the B.S. degree in computer engineering with Boğaziçi University, Istanbul, Turkey. He is currently an Undergraduate Researcher with the Cognition, Learning and Robotics (CoLoRs) Lab, Boğaziçi University, under the supervision of Associate Professor Emre Ugur. He also has professional industry experience as a Software Engineer, focusing on the development of scalable backend architectures and microservices. His research interests include cognitive robotics, robot learning, computer vision, and machine learning.

EMRE UGUR is an Associate Professor of Computer Engineering at Boğaziçi University, Istanbul, Türkiye. He received his B.Sc., M.Sc., and Ph.D. degrees in Computer Engineering from Middle East Technical University (METU), Ankara, Türkiye. He previously worked at ATR Computational Neuroscience Laboratories in Japan, the University of Innsbruck in Austria, and Osaka University in Japan. He currently leads the Cognition, Learning and Robotics (CoLoRs) Lab at Boğaziçi University. His research interests include cognitive robotics, robot learning, developmental robotics, neuro-symbolic artificial intelligence, symbol emergence, and learning from demonstration.