# SPOTTER: Extending Symbolic Planning Operators through Targeted Reinforcement Learning

Vasanth Sarathy*
Smart Information Flow Technologies
Lexintgon, MA
vsarathy@sift.net

Daniel Kasenberg*
Tufts University.
Medford, MA
dmk@cs.tufts.edu

Shivam Goel
Tufts University.
Medford, MA
shivam.goel@tufts.edu

Jivko Sinapov
Tufts University.
Medford, MA
jivko.sinapov@tufts.edu

Matthias Scheutz
Tufts University.
Medford, MA
matthias.scheutz@tufts.edu

## ABSTRACT

Symbolic planning models allow decision-making agents to sequence actions in arbitrary ways to achieve a variety of goals in dynamic domains. However, they are typically handcrafted and tend to require precise formulations that are not robust to human error. Reinforcement learning (RL) approaches do not require such models, and instead learn domain dynamics by exploring the environment and collecting rewards. However, RL approaches tend to require millions of episodes of experience and often learn policies that are not easily transferable to other tasks. In this paper, we address one aspect of the open problem of integrating these approaches: how can decision-making agents resolve discrepancies in their symbolic planning models while attempting to accomplish goals? We propose an integrated framework named SPOTTER that uses RL to augment and support ("spot") a planning agent by discovering new operators needed by the agent to accomplish goals that are initially unreachable for the agent. SPOTTER outperforms pure-RL approaches while also discovering transferable symbolic knowledge and does not require supervision, successful plan traces or any *a priori* knowledge about the missing planning operator.

## KEYWORDS

Planning, Reinforcement Learning

**ACM Reference Format:**
Vasanth Sarathy, Daniel Kasenberg, Shivam Goel, Jivko Sinapov, and Matthias Scheutz. 2021. SPOTTER: Extending Symbolic Planning Operators through Targeted Reinforcement Learning. In *Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), London, UK, May 3–7, 2021*, IFAAMAS, 10 pages.

## 1 INTRODUCTION

Symbolic planning approaches focus on synthesizing a sequence of operators capable of achieving a desired goal [9]. These approaches rely on an accurate high-level symbolic description of the dynamics of the environment. Such a description affords these approaches the benefit of generalizability and abstraction (the model can be used to complete a variety of tasks), available human knowledge, and interpretability. However, the models are often handcrafted, difficult to design and implement, and require precise formulations that can be sensitive to human error. Reinforcement learning (RL) approaches do not assume the existence of such a domain model, and instead attempt to learn suitable models or control policies by trial-and-error interactions in the environment [28]. However, RL approaches tend to require a substantial amount of training in moderately complex environments. Moreover, it has been difficult to learn abstractions to the level of those used in symbolic planning approaches through low-level reinforcement-based exploration. Integrating RL and symbolic planning is highly desirable, enabling autonomous agents that are robust, resilient and resourceful.

Among the challenges in integrating symbolic planning and RL, we focus here on the problem of how partially specified symbolic models can be extended during task performance. Many real-world robotic and autonomous systems already have preexisting models and programmatic implementations of action hierarchies. These systems are robust to many anticipated situations. We are interested in how these systems can adapt to unanticipated situations, autonomously and with no supervision.

In this paper, we present SPOTTER (Synthesizing Planning Operators through Targeted Exploration and Reinforcement). Unlike other approaches to action model learning, SPOTTER does not have access to successful symbolic traces. Unlike other approaches to learning low-level implementation of symbolic operators, SPOTTER does not know *a priori* what operators to learn. Unlike other approaches which use symbolic knowledge to guide exploration, SPOTTER does not assume the existence of partial plans.

We focus on the case where the agent is faced with a symbolic goal but has neither the necessary symbolic operator to be able to synthesize a plan nor its low-level controller implementation. The agent must invent both. SPOTTER leverages the idea that the agent can proceed by planning alone unless there is a discrepancy between its model of the environment and the environment's dynamics. In our approach, the agent attempts to plan to the goal. When no plan is found, the agent explores its environment using an online explorer looking to reach any state from which it can plan to the goal. When such a state is reached, the agent spawns offline "subgoal" RL learners to learn policies which can consistently achieve those conditions. As the exploration agent acts, the subgoal learners learn from its trajectory in parallel. The subgoal learners

*The first two authors contributed equally.

*Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), U. Endriss, A. Nowé, F. Dignum, A. Lomuscio (eds.), May 3–7, 2021, London, UK.* 

regularly attempt to generate symbolic preconditions from which their candidate operators have high value; if such preconditions can be found, the operator is added with those preconditions into the planning domain. We evaluate the approach with experiments in a gridworld environment in which the agent solves three puzzles involving unlocking doors and moving objects.

In this paper, our contributions are as follows: a framework for integrated RL and symbolic planning; algorithms for solving problems in finite, deterministic domains in which the agent has partial domain knowledge and must reach a seemingly unreachable goal state; and experimental results showing substantial improvements over baseline approaches in terms of cumulative reward, rate of learning and transferable knowledge learned.

### 1.1 Running example: GridWorld

Throughout this paper, we will use as a running example a GridWorld puzzle (Figure 1) which an agent must unlock a door which is blocked by a ball. The agent's planning domain abstracts out the notion of specific grid cells, and so all navigation is framed in terms of going to specific objects. Under this abstraction, no action sequence allows the agent to "move the ball out of the way". Planning actions are implemented as handcrafted programs that navigate the puzzle and execute low-level actions (up, down, turn left/right, pickup). Because the door is blocked, the agent cannot generate a symbolic plan to solve the problem; it must synthesize new symbolic actions or operators. An agent using SPOTTER performs RL on low-level actions to learn to reach states from which a plan to the goal exists, and thus can learn a symbolic operator corresponding to "moving the ball out of the way".

## 2 BACKGROUND

In this section, we provide a background of relevant concepts in planning and learning.

### 2.1 Open-World Symbolic Planning

We formalize the planning task as an open-world variant of propositional STRIPS [8], $T = \langle F, O, \sigma_0, \tilde{\sigma}_g \rangle$. We consider $F$ (fluents) to be the set of propositional state variables $f$. A fluent state $\sigma$ is a complete assignment of values to all fluents in $F$. That is, $|\sigma| = |F|$, and $\sigma$ includes positive literals ($f$) and negative literals ($\neg f$). $\sigma_0$ represents the initial fluent state. We assume full observability of the initial fluent state. We can define a partial fluent state $\tilde{\sigma}$ to refer to a partial assignment of values to the fluents $F$. The goal condition is represented as a partial fluent state $\tilde{\sigma}_g$. We define $\mathcal{L}(F)$ to be the set of all partial fluent states with fluents in $F$.

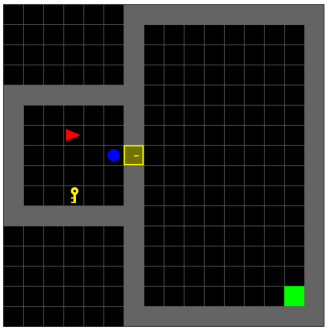


**Figure 1: The agent's (red triangle) goal is to unlock the door (yellow square). SPOTTER can learn how to move the blue ball out of the way. The learned representation is symbolic and can be used to plan (without additional learning) to achieve different goals like reaching the green square.**

The operators under this formalism are open-world. A partial planning operator can be defined as $o = \langle pre(o), \mathit{eff}(o), static(o) \rangle$. $pre(o) \in \mathcal{L}(F)$ are the preconditions, and $\mathit{eff}(o) \in \mathcal{L}(F)$ are the effects, $static(o) \subseteq F$ are those fluents whose values are *known* not to change during the execution of the operator. An operator $o$ is *applicable* in a partial fluent state $\tilde{\sigma}$ if $pre(o) \subseteq \tilde{\sigma}$. The result of executing an operator $o$ from a partial fluent state $\tilde{\sigma}$ is given by the *successor* function $\delta(\tilde{\sigma}, o) = \mathit{eff}(o) \cup restrict(\tilde{\sigma}, static(o))$, where $restrict(\tilde{\sigma}, F')$ is the partial fluent state consisting only of $\tilde{\sigma}$'s values for fluents in $F'$. The set of all partial fluent states defined through repeated application of the successor function beginning at $\tilde{\sigma}$ provides the set of reachable partial fluent states $\Delta_{\tilde{\sigma}}$. A complete operator is an operator $\tilde{o}$ where all the fluents are fully accounted for, namely, $\forall f \in F, f \in \mathit{eff}^+(\tilde{o}) \cup \mathit{eff}^-(\tilde{o}) \cup static(\tilde{o})$, where $\mathit{eff}^+(o) = \{f \in F : f \in \mathit{eff}(o)\}$ and $\mathit{eff}^-(o) = \{f \in F : \neg f \in \mathit{eff}(o)\}$. We assume all operators $o$ satisfy $\mathit{eff}(o) \backslash pre(o) \neq \emptyset$; these are the only operators useful for our purposes.[1] A plan $\pi_T$ is a sequence of operators $\langle o_1, \ldots, o_n \rangle$. A plan $\pi_T$ is executable in state $\sigma_0$ if, for all $i \in \{1, \cdots, n\}$, $pre(o_i) \subseteq \tilde{\sigma}_{i-1}$ where $\tilde{\sigma}_i = \delta(\tilde{\sigma}_{i-1}, o_i)$. A plan $\pi_T$ is said to solve the task $T$ if executing $\pi_T$ from $\sigma_0$ induces a trajectory $\langle \sigma_0, o_1, \tilde{\sigma}_1, \ldots, o_n, \tilde{\sigma}_n \rangle$ that reaches the goal state, namely $\tilde{\sigma}_g \subseteq \tilde{\sigma}_n$.

An open-world forward search (OWFS) is a breadth-first plan search procedure where each node is a partial fluent state $\tilde{\sigma}$. The successor relationships and applicability of $O$ are specified as defined above and used to generate successor nodes in the search space. A plan is synthesized once $\tilde{\sigma}_g$ has been reached. If $\sigma_0$ is a complete state and the operators are complete operators, then each node in the search tree will be complete fluent states, as well.

We also define the notion of a "relevant" operator and "regressor" nodes, as used in backward planning search [9]. For a partial fluent state $\tilde{\sigma}$ with fluents $f_i$ and their valuations $c_i \in \{True, False\}$, operator $o$ is *relevant* at partial fluent state $\tilde{\sigma}$ when:

(1) $restrict(\tilde{\sigma} \backslash \mathit{eff}(o), static(o)) = \tilde{\sigma} \backslash \mathit{eff}(o)$ and
(2) $\tilde{\sigma} \supseteq \mathit{eff}(o) \cup restrict(pre(o), static(o))$

When a relevant operator $o$ is found for a particular $\tilde{\sigma}$, it can be *regressed* to generate a partial fluent state $\delta^{-1}(\tilde{\sigma}, o) = pre(o) \cup (\tilde{\sigma} \backslash \mathit{eff}(o))$.[2] Regression is the "inverse" of operator application in that if applying any operator sequence $\langle o_1, \ldots, o_n \rangle$ yields final state $\tilde{\sigma}'$, if we let $\tilde{\sigma}'' = \delta^{-1}(\ldots(\delta^{-1}(\tilde{\sigma}, o_n)\ldots), o_1)$, then $\tilde{\sigma}'' \subseteq \tilde{\sigma}$. In particular, $\tilde{\sigma}''$ is the *minimal* partial fluent state from which application of $\langle o_1, \cdots, o_n \rangle$ results in $\tilde{\sigma}'$.

[1] We also assume $(\mathit{eff}^+(o) \cup \mathit{eff}^-(o)) \cap static(o) = \emptyset$; i.e. that postconditions and static variables do not conflict.
[2] This is a slight abuse of notation because $\delta^{-1}$ is not the function inverse of $\delta$, but they can be thought of as inverse in the sense described in this paragraph.

## 2.2 Reinforcement Learning

We formalize the environment in which the agent acts as a Markov Decision Process (MDP) $M = \langle S, A, p, r, \iota, \gamma \rangle$, where $S$ is the set of states, $A$ is the set of actions, $p$ is the probability distribution $p(s_{t+1} \mid s_t, a_t)$, $r : S \times A \times S \to \mathbb{R}$ is a reward function, $\iota$ is a probability distribution over initial states, and $\gamma \in (0, 1]$ [28]. A policy for $M$ is defined as the probability distribution $\pi_M(a \mid s)$ that establishes the probability of an agent taking an action $a$ given that it is in the current state $s$. We define the set of all such policies in $M$ as $\Pi_M$. We let $S_0 = \{s \in S : \iota(s) > 0\}$. An RL problem typically consists of finding an optimal policy $\pi^*_M \in \Pi_M$ that maximizes the expected discounted future rewards obtained from $s \in S$:

$$\pi^*_M = \arg\max_{\pi_M} \sum_{s \in S} v_{\pi_M(s)},$$

where $v_{\pi_M(s)}$ is the value function and captures the expected discounted future rewards obtained when starting at state $s$ and selecting actions according to the policy $\pi_M$:

$$v_{\pi_M(s)} = \mathbb{E}_{\pi_M}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right].$$

At each time step, the agent executes an action $a$ and the environment returns the next state $s' \in S$ (sampled from p) and an immediate reward $r$. The experience is then used by the agent to learn and improve its current policy $\pi_M$.

Q-learning [31] is one learning technique in which an agent uses experiences to estimate the optimal Q-function $q^*(s, a)$ for every state $s \in S$ and $a \in A$, where $q^*(s, a)$ is the expected discounted sum of future rewards received by performing action $a$ in state $s$. The Q-function is updated as follows:

$$q(s, a) \leftarrow q(s, a) + \alpha\left[\left(r + \gamma \max_{a' \in A} q(s', a') - q(s, a)\right)\right],$$

where $\alpha \in (0, 1]$ is the learning rate. The Q-learner can explore the environment, e.g., by following an $\epsilon$-greedy policy, in which the agent selects a random action with probability $\epsilon$ and otherwise follows an action with the largest $q(s, a)$.

# 3 PROPOSED SPOTTER FRAMEWORK

We begin by introducing a framework for integrating the planning and learning formulations. We define an integrated planning task that enables us to ground symbolic fluents and operators in an MDP, specify goals symbolically, and realize action hierarchies.

We define an *executor* for a given MDP $M = \langle S, A, p, r, \iota, \gamma \rangle$ as a triple $x = \langle I_x, \pi_x, \beta_x \rangle$ where $I_x \subseteq S$ is an initiation set, $\pi_x(a|s_{init}, s)$ is the probability of performing $a$ given that the executor initialized at state $s_{init}$ and the current state is $s$, and $\beta_x(s_{init}, s)$ expresses the probability of terminating $x$ at $s$ given that $x$ was initialized at $s_{init}$. [3] We define $X_M$ as the set of executors for $M$.

*Definition 3.1.* (Integrated Planning Task) We can formally define an *Integrated Planning Task* (IPT) as $\mathcal{T} = \langle T, M, d, e \rangle$ where $T = \langle F, O, \sigma_0, \tilde{\sigma}_g \rangle$ is an open-world STRIPS task, $M = \langle S, A, p, r, \iota, \gamma \rangle$ is an MDP, a detector function $d : S \mapsto \mathcal{L}(F)$ determines a fluent state for a given MDP state, and an executor function $e : O \mapsto X_M$ that determines a mapping between an operator and an executor in the underlying MDP.

For the purposes of this paper, we assume that for each operator $o \in O$, $e(o)$ is *accurate* to $o$; that is, for every $o \in O$, $I_{e(o)} \supseteq \{s \in S : d(s) \supseteq pre(o)\}$ and

$$\beta_{e(o)}(s_{init}, s) = \begin{cases} 1 & \text{if } d(s) \supseteq \mathit{eff}(o) \\ & \quad \cup \mathit{restrict}(d(s_{init}), \mathit{static}(o)) \\ 0 & \text{otherwise.} \end{cases}$$

The objective of this task is to find an executor $x^\star \in X_M$ such that $\beta_{x^\star}(s_{init}, s) = 1$ if and only if $d(s) \supseteq \tilde{\sigma}_g$, and $I_{x^\star} \supseteq S_0$, and $x^\star$ terminates in finite time.

A *solution* to a particular IPT $\mathcal{T}$ is an executor $x^\star \in X_M$ having the properties defined above.

A *planning solution* to a particular IPT $\mathcal{T}$ is a mapping $\pi_T : S_0 \to O^*$ such that for every $s_0 \in S_0$, $\pi_T(s_0) = \langle o_1, \ldots, o_n \rangle$ is executable at $s_0$ and achieves goal state $\tilde{\sigma}_g$. Assuming all operators are accurate, executing in the MDP $M$ the corresponding executors $e(o_1), \ldots, e(o_n)$ in sequence will yield a final state $s$ such that $d(s) \supseteq \tilde{\sigma}_g$ as desired.

An IPT $\mathcal{T}$ is said to be solvable if a solution exists. It is said to be plannable if a planning solution exists.

## 3.1 The Operator Discovery Problem

As we noted earlier, symbolic planning domains can be sensitive to human errors. One common error is when the domain is missing an operator, which then prevents the agent from synthesizing a plan that requires such an operator. We define a stretch-Integrated Planning Task, stretch-IPT[4], that captures difficult but achievable goals – those for which missing operators must be discovered.

*Definition 3.2.* (Stretch-IPT). A Stretch-IPT $\tilde{\mathcal{T}}$ is an IPT $\mathcal{T}$ for which a solution exists, but a planning solution does not.

Sarathy et al. considered something similar in a purely symbolic planning context as MacGyver Problems [24]; here we extend these ideas to integrated symbolic-RL domains. A planning solution is desirable because a plannable task affords the agent robustness to variations in goal descriptions and a certain degree of generality and ability to transfer decision-making capabilities across tasks. We are interested in turning a stretch-IPT into an plannable IPT, and specifically study how an agent can automatically extend its task description and executor function to invent new operators and their implementations.

*Definition 3.3.* (Operator Discovery Problem). Given a stretch-IPT $\tilde{\mathcal{T}} = \langle T, M, d, e \rangle$ with $T = \langle F, O, \sigma_0, \tilde{\sigma}_g \rangle$, construct a set of operators $O' = \{o'_1, \cdots, o'_m\}$ and their executors $x_{o'_1}, \cdots, x_{o'_m} \in X$ such that the IPT $\langle T', M, d, e' \rangle$ is plannable, with $T' = \langle F, O \cup O', \sigma_0, \tilde{\sigma}_g \rangle$ and the executor function

$$e'(o) = \begin{cases} e(o) & \text{if } o \in O \\ x_o & \text{if } o \in O' \end{cases}.$$

In the rest of the section, we will outline an approach for solving the operator discovery problem.

[3] An executor is simply an option [29] where the policy and termination condition depend on where it was initialized.

[4] akin to "stretch goals" in business productivity

*3.1.1 The Operator Discovery Problem in GridWorld.* In the example GridWorld puzzle, the SPOTTER agent is equipped with a high-level planning domain specified in an open-world extension of PDDL that can be grounded down into an open-world STRIPS problem. This domain is an abstraction of the environment which ignores the absolute positions of objects. In particular, the core movement actions *forward*, *turnLeft*, and *turnRight* in the MDP are absent from the planning domain, which navigates in terms of objects using, e.g., the operator $goToObj(agent, object)$, with preconditions $\neg holding(agent, object)$, $\neg blocked(object)$, and $inRoom(agent, object)$, and with effect $nextToFacing(agent, object)$. All initial operators are assumed to satisfy the closed-world assumptions, except that putting down an object ($putDown(agent, object)$) leaves unknown whether at the conclusion of the action, some other object (e.g., a door) will be blocked. Each operator has a corresponding hand-coded executor. The goal here is $\tilde{\sigma}_g = \{open(door)\}$, which is not achievable using planning alone from the initial state.

## 3.2 Planning and Execution

Our overall agent algorithm is given by Algorithm 1. We consider the agent to be done when it has established that the input IPT $\mathcal{T}$ is a plannable IPT. This is true once the agent is able to find a complete solution to the IPT through planning and execution, alone, and without requiring any learning. Algorithm 1 shows that the agent repeatedly learns in the environment until it has completed a run in which there were no planning impasses, i.e., situations where no plan was found through a forward search in the symbolic space. The set of learners $L$ is maintained between runs of **SOLVE**.

Two sets of fluent states will be important in Algorithms 2-4: $\Sigma_{reach}$ and $\Sigma_{plan}$. $\Sigma_{reach}$ contains all states known to be reachable from the initial state via planning. $\Sigma_{plan}$ contains all states from which a plan to the goal $\tilde{\sigma}_g$ is known.

Algorithm 2 (**SOLVE**) begins with the agent performing an OWFS (open-world forward search) of the symbolic space specified by the task. If a plan is found (line 7), the agent attempts to execute the plan (line 9). If unexpected states are encountered such that the agent cannot perform the next operator, it will call Algorithm 3 (**LEARN**). Otherwise, the agent continues with the next operator until all the operators in a plan are complete. If the goal conditions are satisfied (line 16), the algorithm returns success. Otherwise, it turns to **LEARN**.

## 3.3 Learning Operator Policies

Broadly, in Algorithm 3 (**LEARN**), the agent follows an exploration policy $\pi_{expl}$, e.g., $\epsilon$-greedy, to explore the environment, spawning

**Algorithm 1: SPOTTER($\mathcal{T}$)**

**Input:** $\mathcal{T}$: Integrated Planning Task
1: $impasse \leftarrow true$
2: $L \leftarrow \{\ell_{expl}\}$
3: **while** *impasse* **do**
4: $s_0 \sim T.M.\iota(\cdot)$
5: $impasse, L \leftarrow$ **SOLVE**($\mathcal{T}$, $s_0$, $\emptyset$, $\emptyset$, $\tau$, *false*, $L$))
6: **end while**
7: **return** $\mathcal{T}$

**Algorithm 2: SOLVE($\mathcal{T}$, $s$, $\Sigma_{reach}$, $\Sigma_{plan}$, $\tau$, *impasse*, $L$)**

**Input:** $\mathcal{T}$: Integrated Planning Task
**Input:** $s$: Initial MDP state from which to plan
**Input:** $\Sigma_{reach}$: Set of reachable fluent states
**Input:** $\Sigma_{plan}$: Set of plannable fluent states
**Input:** $\tau$: Value threshold parameter
**Input:** *impasse*: *true* if this algorithm was called from **LEARN**
**Input:** $L$: A set of learners
1: $\sigma \leftarrow T.d(s)$
2: **if** $\mathcal{T}.\tilde{\sigma}_g \subseteq \sigma$ **then**
3: **return** *impasse*, $L$
4: **end if**
5: $\pi_T$, *visitedNodes* $\leftarrow$ OWFS($\mathcal{T}.T, \sigma$)
6: $\Sigma_{reach}$.add( *visitedNodes*)
7: **if** $\pi_T \neq \emptyset$ **then**
8: $\Sigma_{plan}$.add(all *visitedNodes* along $\pi_T$)
9: **for** operator $o_i$ in $\pi_T$ **do**
10: $s \leftarrow$ EXECUTE($\mathcal{T}.e(o_i), s$)
11: $\sigma \leftarrow \mathcal{T}.d(s)$
12: **if** $pre(o_{i+1}) \notin \sigma$ **then**
13: **return** **LEARN**($\mathcal{T}$, $s$, $\Sigma_{reach}$, $\Sigma_{plan}$, $\tau$, $L$)
14: **end if**
15: **end for**
16: **if** $\mathcal{T}.\tilde{\sigma}_g \subseteq \sigma$ **then**
17: **return** *impasse*, $L$
18: **else**
19: **return** **LEARN**($\mathcal{T}$, $s$, $\Sigma_{reach}$, $\Sigma_{plan}$, $\tau$, $L$)
20: **end if**
21: **else**
22: **return** **LEARN**($\mathcal{T}$, $s$, $\Sigma_{reach}$, $\Sigma_{plan}$, $\tau$, $L$)
23: **end if**

RL agents which attempt to construct policies to reach particular partial fluent states from which the goal $\tilde{\sigma}_g$ is reachable by planning. These partial fluent states correspond to operator effects. For each such set of effects, the system attempts to find sets of preconditions from which that operator can consistently achieve high value; when one such set of preconditions is found, the policy along with the corresponding set of preconditions and effects is used to define a new operator and its corresponding executor, which are added to the IPT.

Algorithm receives as input, among other things, a set of learners $L$ (which may grow during execution, as described below). $L$ includes an exploration agent $\ell_{expl}$ with corresponding policy $\pi_{expl}$ (initialized in Algorithm 1). $L$ also contains a set of offline "subgoal" learners, which initially is empty.

In each time step, **LEARN** executes an action $a$ according to its exploration policy $\pi_{expl}$, with resulting MDP state $s'$ and corresponding fluent state $\sigma$ (lines 4-6). The agent will attempt to check if $\sigma$ is a fluent state from which it can plan to the goal. First the agent checks if $\sigma$ is already *known* to be a state from which it can plan to the goal ($\sigma \in \Sigma_{plan}$; line 7). If not, the agent attempts to plan from $\sigma$ to the goal (line 10). If there is a plan, then the agent regresses each operator in the plan in reverse order ($o_n, \cdots, o_1$). As described in Section 2.1, after regressing through $o_i$, the resulting fluent state $\tilde{\sigma}_{i-1}$ is the most general possible partial fluent state (i.e., containing the least possible fluents) such that executing $\langle o_i, \cdots, o_n \rangle$ from $\tilde{\sigma}_{i-1}$ results in $\tilde{\sigma}_g$; this makes each $\tilde{\sigma}_{i-1}$ a prime candidate for some

new operator $o^\star$'s effects. That is, assuming $static(o) = \emptyset$, $\tilde{\sigma}_{i-1}$ guarantees that $\langle o_i, \cdots, o_n \rangle$ is a plan to $\tilde{\sigma}_g$, while allowing the corresponding executor to terminate in the largest possible set of fluent states. Each such partial fluent state $\tilde{\sigma}_{sg}$ is chosen as a subgoal, for which a new learner $\ell_{\tilde{\sigma}_{sg}}$ is spawned and added to $L$ (lines 12-18). $\tilde{\sigma}_{sg}$ is also added to the set of "plannable" fluent states $\Sigma_{plan}$.

Each subgoal learner $\ell_{\tilde{\sigma}_{sg}}$ is trained each time step using as reward the indicator function $\mathbf{1}_{\sigma \supseteq \tilde{\sigma}_{sg}}$, which returns 1 if that learner's subgoal is satisfied by $s'$ and 0 otherwise (lines 23-25).

While in this case the exploration learner $\ell_{expl}$ may be conceived as an RL agent whose reward function is 1 whenever *any* such subgoal is achieved (line 22) and whose exploration policy $\pi_{expl}$ is $\epsilon$-greedy, nothing prevents it from being defined differently as a random agent or even symbolic learner as discussed in Section 6.

**Algorithm 3:** LEARN$(\mathcal{T}, s, \Sigma_{reach}, \Sigma_{plan}, \tau, L)$

**Input:** $\mathcal{T}$: Integrated Planning Task
**Input:** $s$: Initial MDP state
**Input:** $\Sigma_{reach}$: Set of reachable fluent states
**Input:** $\Sigma_{plan}$: Set of plannable fluent states
**Input:** $\tau$: Value threshold
**Input:** $L$: set of learners

1: $done \leftarrow false$
2: $\sigma \leftarrow T.d(s)$
3: **while** $\neg done$ **do**
4: $\quad a \sim \pi_{expl}(\cdot \mid s)$
5: $\quad s' \sim T.p(\cdot \mid s, a)$
6: $\quad \sigma \leftarrow \mathcal{T}.d(s')$
7: $\quad$ **if** $\sigma \supseteq \tilde{\sigma}$ for some $\tilde{\sigma} \in \Sigma_{plan}$ **then**
8: $\quad\quad done \leftarrow true$
9: $\quad$ **else**
10: $\quad\quad \pi_T, visitedNodes \leftarrow$ OWFS$(\mathcal{T}.T, \sigma)$
11: $\quad\quad$ **if** $\pi_T \neq \emptyset$ **then**
12: $\quad\quad\quad \tilde{\sigma}_{sg} \leftarrow \tilde{\sigma}_g$
13: $\quad\quad\quad$ **for** $o_i$ in reversed$(\pi_T)$ **do**
14: $\quad\quad\quad\quad \tilde{\sigma}_{sg} \leftarrow \delta^{-1}(\tilde{\sigma}_{sg}, o_i)$
15: $\quad\quad\quad\quad \Sigma_{plan}$.add$(\tilde{\sigma}_{sg})$
16: $\quad\quad\quad\quad \ell_{\tilde{\sigma}_{sg}} \leftarrow$ spawnLearner$(\tilde{\sigma}_{sg})$
17: $\quad\quad\quad\quad L \leftarrow L \cup \{\ell_{\tilde{\sigma}_{sg}}\}$
18: $\quad\quad\quad$ **end for**
19: $\quad\quad\quad done \leftarrow true$
20: $\quad\quad$ **end if**
21: $\quad$ **end if**
22: $\quad \ell_{expl}$.train$(s, a, \mathbf{1}_{done=true}, s')$
23: $\quad$ **for** $\ell_{\tilde{\sigma}_{sg}} \in L$ **do**
24: $\quad\quad \ell_{\tilde{\sigma}_{sg}}$.train$(s, a, \mathbf{1}_{\sigma \supseteq \tilde{\sigma}_{sg}}, s')$
25: $\quad$ **end for**
26: $\quad s \leftarrow s'$
27: **end while**
28: **for** $\ell_{\tilde{\sigma}_{sg}} \in L$ **do**
29: $\quad$ **for** $\tilde{\sigma}_{pre} \in$ GEN-PRECON$(\ell_{\tilde{\sigma}_{sg}}, \Sigma_{reach}, \tau)$ **do**
30: $\quad\quad o^\star \leftarrow \langle \tilde{\sigma}_{pre}, \tilde{\sigma}_{sg}, \emptyset \rangle$
31: $\quad\quad x^\star \leftarrow$ makeExecutor$(\ell_{\tilde{\sigma}_{sg}}, o^\star)$
32: $\quad\quad \mathcal{T}$.addOperator$(o^*, x^\star)$
33: $\quad$ **end for**
34: **end for**
35: **return** SOLVE$(\mathcal{T}, s, \Sigma_{reach}, \Sigma_{plan}, \tau, true, L)$

Upon reaching a state from which a plan to the goal exists, the agent stops exploring. For each of its subgoal learners $\ell_{\tilde{\sigma}_{sg}} \in L$, it attempts to construct sets of preconditions (characterized as a partial fluent state $\tilde{\sigma}_{pre}$) from which its policy can consistently achieve the subgoal state $\tilde{\sigma}_{sg}$ (see Section 3.4) (lines 28-34). If any such precondition sets $\tilde{\sigma}_{pre}$ exist, the agent constructs the operator $o^\star$ such that $pre(o^\star) = \tilde{\sigma}_{pre}$, $eff(o^\star) = \tilde{\sigma}_{sg}$, and without static fluents (all other variables are unknown once the operator is executed; $static(o^\star) = \emptyset$). The corresponding executor is constructed as makeExecutor$(\ell_{\tilde{\sigma}_{sg}}, o^\star) = \langle I_{x^\star}, \pi_{x^\star}, \beta_{x^\star} \rangle$, where

$$I_{x^\star} = \{s' \in S : T.d(s') \supseteq pre(o^\star)\}$$

$$\pi_{x^\star}(a \mid s_{init}, s) = \begin{cases} 1 & \text{if } a = \underset{a' \in T.M.A}{\arg\max}\ \ell_{\tilde{\sigma}_{sg}}.q(s, a) \\ 0 & \text{otherwise} \end{cases}$$

$$\beta_{x^\star}(s_{init}, s) = \begin{cases} 1 & \text{if } T.d(s) \supseteq \tilde{\sigma}_{sg} \\ 0 & \text{otherwise.} \end{cases}$$

Note that while the definition of the Operator Discovery Problem allows the constructed operators' executors to depend on $s_{init}$, in practice the operators are ordinary options. Options are sufficient, *provided the operators being constructed have no static fluents.*[5] Constructing operators with static fluents is a topic for future work.

Constructed operators and their corresponding executors are added to the IPT (line 32), which ideally becomes plannable as a result, solving the Operator Discovery Problem. Because the agent is currently in a state from which it can plan to the goal, control is passed back to SOLVE (with *impasse* set to *true* because this episode required learning).

*3.3.1 Learning operator policies in GridWorld.* The puzzle shown in Figure 1 presents a Stretch-IPT for which a solution exists in terms of the MDP-level actions, but no planning solution exists. Thus, SPOTTER enters LEARN, and initially moves around randomly (the exploration policy has not yet received a reward). Eventually, in the course of random action, the agent moves the ball out of the way (Algorithm LEARN, line 11). From this state, the action plan

$$\pi_T = goToObj(agent, key); pickUp(agent, key); \\ goToObj(agent, door); useKey(agent, door)$$

achieves the goal $\tilde{\sigma}_g = \{open(door)\}$. Regressing through this plan from the goal state (Algorithm LEARN, lines 13-17), the agent identifies the subgoal

$$\tilde{\sigma}_{sg} = \{nextToFacing(agent, ball), \\ handsFree(agent), inRoom(agent, key), \\ inRoom(agent, door), locked(door), \\ \neg holding(agent, key), \neg blocked(door)\}$$

(as well as other subgoals corresponding to the suffixes of $\pi_T$), and constructs a corresponding subgoal learner, $\ell_{\tilde{\sigma}_{sg}}$ which uses RL to learn a policy which consistently achieves $\tilde{\sigma}_{sg}$.

[5] This also indicates why we used our open-world planning formalism: in closed-world planning, all fluents not in the effects are static, requiring executors which depend on $s_{init}$. By specifying static fluents for each operator, we can leverage handmade operators which (largely) satisfy the closed-world assumption while allowing the operators returned by SPOTTER to be open-world.

## 3.4 Generating Preconditions

The prior algorithms allow an agent to plan in the symbolic space, and when stuck explore a subsymbolic space with RL learners. We noted that each learner is connected to a particular subgoal – from which the agent can generate a symbolic plan – which in turn, maps onto the effects of a potential new operator. What remains is to define the preconditions of such an operator. Algorithm 4 (GEN-PRECON) incrementally generates increasingly general sets of preconditions for each learner.

GEN-PRECON begins with initializing a set of above-threshold fluent states $\Sigma_{>\tau}$ to empty. This set will represent the output of the algorithm, which in turn is essentially a form of graph search over the space of possible sets of preconditions. More specifically, GEN-PRECON first adds all the fluent states in the set of reachable fluent states $\Sigma_{reach}$ to the search queue and to $\Sigma_{>\tau}$ (lines 3-7). The learner $\ell$ has visited states in the MDP and has been updating its Q-table as part of LEARN (see line 24 in LEARN). We probe this Q-table and extract the fluent states that the agent has visited and populate a set of "been" fluent states $\Sigma_{been}$ (line 9). While there are fluent states (or partial fluent states) in the queue, a set of "successor" nodes is computed for each one (node) by capturing the set of fluents common to both the node and a particular "been" fluent state $\sigma'$, for each $\sigma' \in \Sigma_{been}$. If the average value (according to $\ell$'s $q$ function) of all states satisfying $\tilde{\sigma}_{common}$ is above the threshold $\tau$, then the above-threshold common partial fluent state $\tilde{\sigma}_{common}$ is added to $\Sigma_{>\tau}$. The idea here is to only allow sets of preconditions which are guaranteed reachable by the planner (generalizations of elements of $\Sigma_{reach}$), and to compute the set of all such sets of preconditions for which the agent can consistently achieve the learner's subgoal $\tilde{\sigma}_{sg}$ (where for the purposes of this paper, a set of preconditions $\tilde{\sigma}_{pre}$ "consistently achieves" $\tilde{\sigma}_{sg}$ if the average value of all MDP states satisfying $\tilde{\sigma}_{pre}$ is above the threshold $\tau$).

**Algorithm 4:** GEN-PRECON$(\ell, \Sigma_{reach}, \tau)$

**Input:** $\ell$: Learner along with Q-tables
**Input:** $\Sigma_{reach}$: Fluent states reachable from the initial state
**Input:** $\tau$: Value threshold

1: $\Sigma_{>\tau} \leftarrow \emptyset$
2: $queue \leftarrow \emptyset$
3: **for** $\sigma$ in $\Sigma_{reach}$ **do**
4:  **if** value$(\sigma) > \tau$ **then**
5:  $queue$.add$(\sigma)$
6:  $\Sigma_{>\tau}$.add$(\sigma)$
7:  **end if**
8: **end for**
9: $\Sigma_{been} \leftarrow$ getFluentStatesFromQ$(\ell)$
10: **while** $queue$ **do**
11:  $node \leftarrow queue$.pop()
12:  **for** $\sigma'$ in $\Sigma_{been}$ **do**
13:  $\tilde{\sigma}_{common} \leftarrow node \cap \sigma'$
14:  **if** value$(\tilde{\sigma}_{common}) > \tau \wedge \tilde{\sigma}_{common}$ not in $\Sigma_{>\tau}$ **then**
15:  $queue$.add$(\tilde{\sigma}_{common})$
16:  $\Sigma_{>\tau}$.add$(\tilde{\sigma}_{common})$
17:  **end if**
18:  **end for**
19: **end while**
20: **return** $\Sigma_{>\tau}$

GEN-PRECON can be terminated at any time during the while loop, and will yield zero or more plausible preconditions for a particular learner-operator. If terminated before any nodes have been expanded, GEN-PRECON will simply yield elements from $\Sigma_{reach}$. However, allowing this algorithm to run longer will yield more general preconditions, i.e., those with fewer fluents.

*3.4.1 Precondition generation in GridWorld.* Returning to the puzzle in Figure 1, the precondition generation algorithm uses the value function for the subgoal learner $\ell_{\tilde{\sigma}_{sg}}$ to determine a conjunction of fluents such that the MDP states satisfying those fluents have a high average state value (say, $> 0.9$), and which is plannable from the environment's start state.[6] This conjunction specifies the new operator's preconditions. The operator is added to the planning domain, and this augmented domain can be solved using planning alone, and can be used in other plans to achieve other goals in the environment.

# 4 EXPERIMENTS

We evaluate SPOTTER on MiniGrid [5], a platform based on procedurally generated 2D gridworld environments populated with objects – agent, balls, doors and keys. The agent can navigate the world, manipulate objects, and move between rooms.

To establish the fact that (1) operator discovery is possible, and (2) that the symbolic operators discovered by SPOTTER are useful for performing new tasks in the environment, we structure our evaluation into three puzzles that the agent must solve in sequence, much like three levels in puzzle video games (Fig. 1). The environment in each level consists of two rooms with a locked door separating them. The agent and the key to the room are randomly placed in the leftmost room. In puzzle 1, the agent's task is to pickup the key, and use it to open the door. The episode terminates, and the agent receives a reward inversely proportional to the number of elapsed time steps, when the door is open. In puzzle 2 the agent's goal is again to open the door, but a ball is placed directly in front of the door, which the agent must pick up and move elsewhere before picking up the key (this is the running example). Puzzle 3 is identical to puzzle 2, except that the agent's goal is now not to open the door, but to go to a goal location (green square) in the far corner of the rightmost room. The high-level planning domain and low-level RL actions are as defined for the running example.

The evaluation is structured so that for almost all initial conditions in puzzle 1, SPOTTER's planner can produce a plan that can be successfully executed to reach the goal state. In puzzle 2, the door is blocked by the ball, and the agent has no operator representing "move the ball out of the way" (and no combination of existing operators can safely achieve this effect given the agent's planning domain). The agent must discover such an operator. Finally, puzzle 3 is designed to test whether the learned operator can be used in the execution of different goals in the same environment.

Figure 2 shows the average results of running SPOTTER 10 times on puzzles 1 through 3. The algorithm was allowed to retain any operators/executors discovered between puzzles 2 and 3. In each case we employed a constant learning rate $\alpha = 0.1$, discount factor $\gamma = 0.99$, and an $\epsilon$-greedy exploration policy with

[6] The preconditions produced are too lengthy to include in this paper, but can be found in the supplementary material.

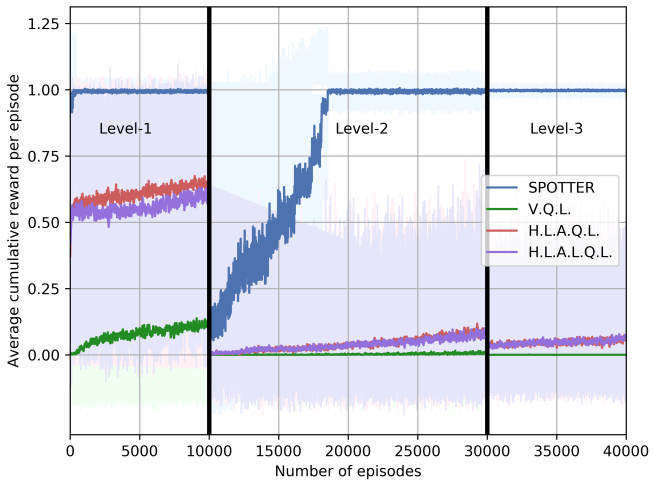



**Figure 2: Experimental performance across three tasks. We report mean cumulative reward (and standard deviation in lighter color) obtained by our approach (SPOTTER) and three baseline algorithms: tabular Q-learning over primitive actions (VQL), tabular Q-learning with primitive and high-level action executors (HLAQL), and HLAQL with q-updates trickling down from HLAs to primitives (HLALQL).**

decaying exploration constant $\epsilon$ beginning at 0.9 and decaying towards 0.05. $\epsilon$ is decayed exponentially using the formula $\epsilon(t) = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min})e^{-\lambda t}$, where $\lambda = -log(0.01)/N$, $N$ being the maximum number of episodes. The value threshold was set to $\tau = 0.9$. We ran for 10,000 episodes on puzzle 1, 20,000 episodes on puzzle 2, and 10,000 episodes on puzzle 3.

The experimental results show that SPOTTER achieved higher overall rewards than the baselines in the given time frame and did so more quickly.[7] Crucially, the agent learned the missing operator for moving the blue ball out of the way in Level 2, and was immediately able to use this operator in Level 3. This is demonstrated both by the fact that the agent did not experience any drop in performance when transitioning to Level 3 and also we know from running the experiment that the agent did not enter **LEARN** or **GEN-PRECON** in Level 3. It is important to note that the baselines did converge at around 800,000 - 1,000,000 episodes, significantly later than SPOTTER.[8].

The results suggest that SPOTTER significantly outperformed comparable baselines. The HLAQL and HLALQL baselines have their action space augmented with the same high-level executors provided to SPOTTER. SPOTTER does not use any function approximation techniques and the exploration and subgoal learners are tabular Q-learners themselves. Accordingly, we did not compare against any deep RL baselines. We also did not compare transfer learning and curriculum learning approaches as these approaches do not handle cases where new representations need to be learned from the environment.

[7]Code implementing SPOTTER and the baselines along with experiments will be made available post-review.

[8]In the supplementary material, we provide the learned operator described in PDDL, learning curves for the baselines over 2,000,000 episodes, and videos showing SPOTTER's integrated planning and learning.

## 4.1 Experiment 2

Ordinarily, as soon as SPOTTER discovers an operator exceeding the value threshold, that operator is incorporated into the planning agent's model. We dispensed with this assumption and ran SPOTTER on puzzle 2 for 50,000 episodes, allowing the system to continue learning operator policies and generating preconditions throughout this time. Every 50 episodes, SPOTTER logged the new operators it had created. (Operators were not logged if their preconditions were specifications of preconditions for which an operator had already been discovered.) Figure 3 shows the results for one particular operator learner (i.e., each of the output operators has the same postconditions and the same underlying policy, but has a unique set of preconditions). As Figure 3a indicates, by episode 29,500 this learner discovered 9,049 unique operators; no further operators were discovered after this episode.

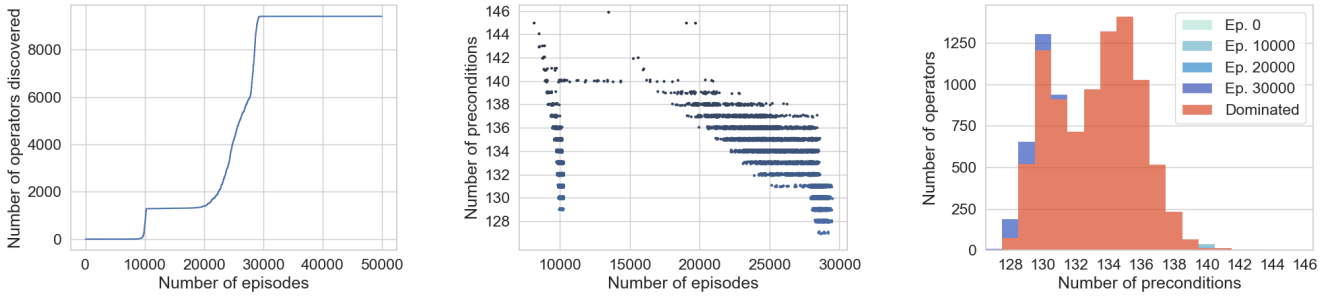
Figures 3b and 3c demonstrate that with additional exploration, the agent can construct more general operators. Recall that an operator with a set of preconditions is accepted whenever the average value of all MDP states satisfying those preconditions is greater than the value threshold (in this experiment, 0.9). As the values of additional MDP states increase past the threshold, more general operators (with fewer preconditions) cross this threshold. Figure 3b plots, for each operator discovered, the episode in which it was logged and its total number of preconditions. Note that there are several "waves" of precondition generalization. Beginning with the discovery of the first viable set of preconditions, as additional states cross the threshold there is a rapid discovery of new operators with additional preconditions. Eventually (a little after 10,000 episodes), the existing operators are sufficiently general that many rarely-seen MDP states would have to be thoroughly explored before more general preconditions can be found. For a while, any new operators discovered (while they are not merely specifications of existing operators) have a larger number of preconditions. This ultimately culminates in a "second wave" in which enough MDP states have been explored that more general operators can be produced, ultimately surpassing the first wave.

Figure 3c shows that operators created later not only have fewer preconditions than earlier operators, but *dominate* earlier operators (in that the preconditions of the dominated operator are a strict superset of the preconditions of the dominating operator). Orange bars represent dominated operators, blue bars those for which a superior operator has not yet been found. Relatively few non-dominated operators persist by step 30,000, and nearly all that do were created in the last few thousand episodes, suggesting that running SPOTTER for longer before incorporating operators allows the construction of strictly better (more general) operators.[9]

## 5 DISCUSSION

The SPOTTER architecture assumes the existence of a planning domain with operators which correspond to executors which are more

[9]An animated version of this chart, showing how operators are constructed and then dominated as the agent continues exploring, appears in the supplementary material.




**(a) The number of unique sets of preconditions discovered for which average value is above threshold increases as SPOTTER is allowed to continue exploring.**

**(b) Generation of operators with decreasing numbers of preconditions proceeds in "waves" as the value estimates of additional MDP states converge.**

**(c) By episode 30,000, almost all operators have been dominated (replaced by superior operators), and almost all non-dominated operators were created late in the run.**

**Figure 3: Results of precondition generalization experiment for a single subgoal learner on puzzle 2.**

or less correct. It does *not* assume that these executors are reward-optimal. Further, some tasks can be more efficiently performed if they are not first split into subgoals. Thus, the performance of raw RL systems eventually overtakes that of SPOTTER on any particular environment. This is not a serious flaw – SPOTTER also produces knowledge that can be more easily applied to perform other tasks in the same environment.

The environments used to test SPOTTER (puzzles 1 through 3) are deterministic. In stochastic environments, it is often difficult to design planning domains where operators have guaranteed effects. While SPOTTER can handle stochastic environments, it would need more robust metrics for assessing operator confidence.

Future work could also emphasize adapting this work to high-dimensional continuous state and action spaces using deep reinforcement learning. "Symbolizing" such spaces can be difficult, and in particular such work would have to rethink how to generate candidate preconditions, since the existing approach enumerates over all states, which clearly would not work in deep RL domains.

The key advantage of SPOTTER is that the agent can produce operators that can potentially be applied to other tasks in the same environment. Because these operators' executors are policies (here, policies over finite, atomic MDPs), they do not generalize particularly well to environments with different dynamics or unseen start states (e.g., an operator learned in puzzle 2 could not possibly be applied in puzzle 3 if the door was moved up or down by even one cell). While function approximation could be helpful to solving this problem, an ideal approach might be a form of program synthesis [27], in which the agent learns *programs* that can be applied regardless of environment.

Furthermore, in this work, we manually sequenced the three tasks to elicit the discovery of operators that would be useful in the final task environment. A possible avenue for future work would be to develop automated task sequencing methods (i.e., curriculum learning [22]) as to improve performance in downstream tasks.

## 6 RELATED WORK

In this section, we review related work that overlaps with various aspects of the proposed integrated planning and learning literature.

### 6.1 Learning Symbolic Action Models

Early work in the symbolic AI literature explored how agents can adjust and improve their symbolic models through experimentation. Gill proposed a method for learning by experimentation in which the agent can improve its domain knowledge by finding missing operators [10]. The agent is able design experiments at the symbolic level based on observing the symbolic fluent states and comparing against an operator's preconditions and effects. Other approaches (to name a few: [11, 17, 21, 25, 26, 30]), comprising a significant body of literature, have explored recovering from planning failures, refining domain models, and open-world planning.

These approaches do not handle the synthesis of the underlying implementation of an operator as we have proposed. That is, they synthesize new operators, but no executors. That said, the rich techniques offered by these approaches can be useful in integrating into our online learner. As we describe in Section 3.3, the online learner follows an $\epsilon$-greedy exploration policy. Future work will explore how our online learner could be extended to conduct symbolically-guided experiments as these approaches suggest.

More recently, there has been a growing body of literature exploring how domain models can be learned from action traces [3, 14, 16, 36]. The ARMS system learns PDDL planning operators from examples. The examples consist of successful fluent state and operator pairs corresponding to a sequence of transitions in the symbolic domain [35]. Subsequent work has explored how domains can be learned with partial knowledge of successful traces [2, 6], and with neural networks capable of approximating from partial traces [33] and learning models from pixels [4, 7].

In the RL context, there has been recent work in learning symbolic concepts [18] from low-level actions. Specifically, Konidaris et al. assume the agent has available to it a set of high-level actions, couched in the options framework for hierarchical reinforcement learning. The agent must learn a symbolic planning domain with operators and their preconditions and effects. This approach to integrating planning and learning is, in a sense, a reverse of our approach. While their use case is an agent that has high-level actions but no symbolic representation, ours assumes that we have

(through hypothesizing from the backward search) most of the symbolic scaffolding, but need to learn the policies themselves.

### 6.2 Learning Action Executors

There has been a tradition of research in leveraging planning domain knowledge and symbolically provided constraints to improve the sample efficiency of RL systems. Grzes et al. use a potential function (as a difference between source and destination fluent states) to shape rewards of a Q-learner [13]. While aligned with our own targeted Q-learners, their approach requires the existence of symbolic plans, which our agent does not possess. A related approach (PLANQ) combines STRIPS planning with Q-learning to allow an agent to converge faster to an optimal policy [12]. Planning with Partially Specified Behaviors (PPSB) is based on PLANQ, and takes as input symbolic domain information and produces a Q-values for each of the high-level operators [1]. Like Grzes et al., PLANQ and PPSB assume the existence of a plan, or at least a plannable task. Other recent approaches have extended these ideas to incorporate using partial plans [15]. The approach proffered by Illanes et al. improves the underlying learner's sample efficiency, but can also be provided with symbolic goals. However, their approach assumes the agent has a complete (partial order) plan for its environment, and can use that plan to construct a reward function for learning options to accomplish a task. One key difference between their work and ours is that we construct our own operators, whereas they use a set of existing operators to learn policies.

Generally, while these methods learn action implementations, they assume the existence of a successful plan or operator definition. In the proposed framework, the agent has neither and must hypothesize operators and their corresponding executors.

RL techniques have also been used to improve the quality of symbolic plans. The DARLING framework uses symbolic planning and reasoning to constrain exploration and reduce the search space, and uses RL to improve the quality of the plan produced [19]. While our approach shares the common goal of reducing the brittleness of planning domains, they do not modify the planning model. The PEORL framework works to choose a plan that maximizes a reward function, thereby improving the quality of the plan using RL [34]. More recently Lyu et al. propose a framework (SDRL) for generalizing the PEORL framework with intrinsic goals and integration with a deep reinforcement learning (DRL) machinery [20]. However, neither PEORL nor SDRL synthesizes new operators or learns planning domains. Our work also differs from the majority of model-based RL approaches [23, 32] in that we are interested in STRIPS-like explicit operators that are useful in symbolic planning, and thereby transferable to a wide range of tasks within a domain.

## 7 CONCLUSION

Automatically synthesizing new operators during task performance is a crucial capability needed for symbolic planning systems. As we have examined here, such a capability requires a deep integration between planning and learning, one that benefits from leveraging RL to fill in missing connections between symbolic states. While exploring the MDP state space, SPOTTER can identify states from which symbolic planning is possible, use this to identify subgoals for which operators can be synthesized, and learn policies for these operators by RL. SPOTTER thus allows a symbolic planner to explore previously unreachable states, synthesize new operators, and accomplish new goals.

## 8 ACKNOWLEDGEMENTS

This work was funded in part by NSF grant IIS-2044786 and DARPA grant W911NF-20-2-0006.

## REFERENCES

[1] Javier Segovia Aguas, Jonathan Ferrer-Mestres, and Anders Jonsson. 2016. Planning with Partially Specified Behaviors.. In *CCIA*. 263–272.
[2] Diego Aineto, Sergio Jiménez, and Eva Onaindia. 2019. Learning STRIPS action models with classical planning. *arXiv preprint arXiv:1903.01153* (2019).
[3] Ankuj Arora, Humbert Fiorino, Damien Pellier, Marc Métivier, and Sylvie Pesty. 2018. A review of learning planning action models. *The Knowledge Engineering Review* 33 (2018).
[4] Masataro Asai and Alex Fukunaga. 2017. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. *arXiv preprint arXiv:1705.00154* (2017).
[5] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. 2018. Minimalistic Gridworld Environment for OpenAI Gym. https://github.com/maximecb/gym-minigrid.
[6] Stephen N Cresswell, Thomas Leo McCluskey, and Margaret M West. 2013. Acquiring planning domain models using LOCM. *Knowledge Engineering Review* 28, 2 (2013), 195–213.
[7] Andrea Dittadi, Thomas Bolander, and Ole Winther. 2018. Learning to Plan from Raw Data in Grid-based Games.. In *GCAI*. 54–67.
[8] Richard E Fikes and Nils J Nilsson. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2, 3-4 (1971), 189–208.
[9] Malik Ghallab, Dana Nau, and Paolo Traverso. 2016. *Automated planning and acting*. Cambridge University Press.
[10] Yolanda Gil. 1994. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Machine Learning Proceedings 1994*. Elsevier, 87–95.
[11] Evana Gizzi, Mateo Guaman Castro, and Jivko Sinapov. 2019. Creative Problem Solving by Robots Using Action Primitive Discovery. In *2019 Joint IEEE 9th International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*. IEEE, 228–233.
[12] Matthew Grounds and Daniel Kudenko. 2005. Combining reinforcement learning with symbolic planning. In *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning*. Springer, 75–86.
[13] Marek Grzes and Daniel Kudenko. 2008. Plan-based reward shaping for reinforcement learning. In *2008 4th International IEEE Conference Intelligent Systems*, Vol. 2. IEEE, 10–22.
[14] Chad Hogg, Ugur Kuter, and Hector Munoz-Avila. 2010. Learning methods to generate good plans: Integrating htn learning and reinforcement learning. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*. Citeseer.
[15] León Illanes, Xi Yan, Rodrigo Toro Icarte, and Sheila A McIlraith. 2020. Symbolic Plans as High-Level Instructions for Reinforcement Learning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 30. 540–550.
[16] Sergio Jiménez, Tomás De La Rosa, Susana Fernández, Fernando Fernández, and Daniel Borrajo. 2012. A review of machine learning for automated planning. *The Knowledge Engineering Review* 27, 4 (2012), 433–467.
[17] Saket Joshi, Paul Schermerhorn, Roni Khardon, and Matthias Scheutz. 2012. Abstract planning for reactive robots. In *2012 IEEE International Conference on Robotics and Automation*. IEEE, 4379–4384.
[18] George Konidaris, Leslie Pack Kaelbling, and Tomas Lozano-Perez. 2018. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research* 61 (2018), 215–289.
[19] Matteo Leonetti, Luca Iocchi, and Peter Stone. 2016. A synthesis of automated planning and reinforcement learning for efficient, robust decision-making. *Artificial Intelligence* 241 (2016), 103–130.
[20] Daoming Lyu, Fangkai Yang, Bo Liu, and Steven Gustafson. 2019. SDRL: interpretable and data-efficient deep reinforcement learning leveraging symbolic planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 2970–2977.
[21] Tom M Mitchell, Richard M Keller, and Smadar T Kedar-Cabelli. 1986. Explanation-based generalization: A unifying view. *Machine learning* 1, 1 (1986), 47–80.
[22] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. 2020. Curriculum Learning for Reinforcement Learning Domains: A Framework and Survey. *Journal of Machine Learning Research* 21, 181 (2020), 1–50. http://jmlr.org/papers/v21/20-212.html

[23] Jun Hao Alvin Ng and Ronald PA Petrick. 2019. Incremental Learning of Planning Actions in Model-Based Reinforcement Learning.. In *IJCAI*. 3195–3201.

[24] Vasanth Sarathy and Matthias Scheutz. 2018. MacGyver problems: Ai challenges for testing resourcefulness and creativity. *Advances in Cognitive Systems* 6 (2018), 31–44.

[25] Wei-Min Shen. 1989. *Learning from the environment based on percepts and actions*. Ph.D. Dissertation. Carnegie Mellon University.

[26] Wei-Min Shen and Herbert A Simon. 1989. Rule Creation and Rule Learning Through Environmental Exploration.. In *IJCAI*. Citeseer, 675–680.

[27] Armando Solar-Lezama. 2009. The sketching approach to program synthesis. In *Asian Symposium on Programming Languages and Systems*. Springer, 4–13.

[28] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[29] Richard S Sutton, Doina Precup, and Satinder Singh. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence* 112, 1-2 (1999), 181–211.

[30] Kartik Talamadupula, J Benton, Subbarao Kambhampati, Paul Schermerhorn, and Matthias Scheutz. 2010. Planning for human-robot teaming in open worlds. *ACM Transactions on Intelligent Systems and Technology (TIST)* 1, 2 (2010), 1–24.

[31] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.

[32] John Winder, Stephanie Milani, Matthew Landen, Erebus Oh, Shane Parr, Shawn Squire, Marie desJardins, and Cynthia Matuszek. 2019. Planning with Abstract Learned Models While Learning Transferable Subtasks. *arXiv preprint arXiv:1912.07544* (2019).

[33] Zhanhao Xiao, Hai Wan, Hankui Hankz Zhuo, Jinxia Lin, and Yanan Liu. 2019. Representation Learning for Classical Planning from Partially Observed Traces. *arXiv preprint arXiv:1907.08352* (2019).

[34] Fangkai Yang, Daoming Lyu, Bo Liu, and Steven Gustafson. 2018. Peorl: Integrating symbolic planning and hierarchical reinforcement learning for robust decision-making. *arXiv preprint arXiv:1804.07779* (2018).

[35] Qiang Yang, Kangheng Wu, and Yunfei Jiang. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence* 171, 2-3 (2007), 107–143.

[36] Terry Zimmerman and Subbarao Kambhampati. 2003. Learning-assisted automated planning: looking back, taking stock, going forward. *AI Magazine* 24, 2 (2003), 73–73.