Reward Machines: Exploiting Reward Function Structure in RL

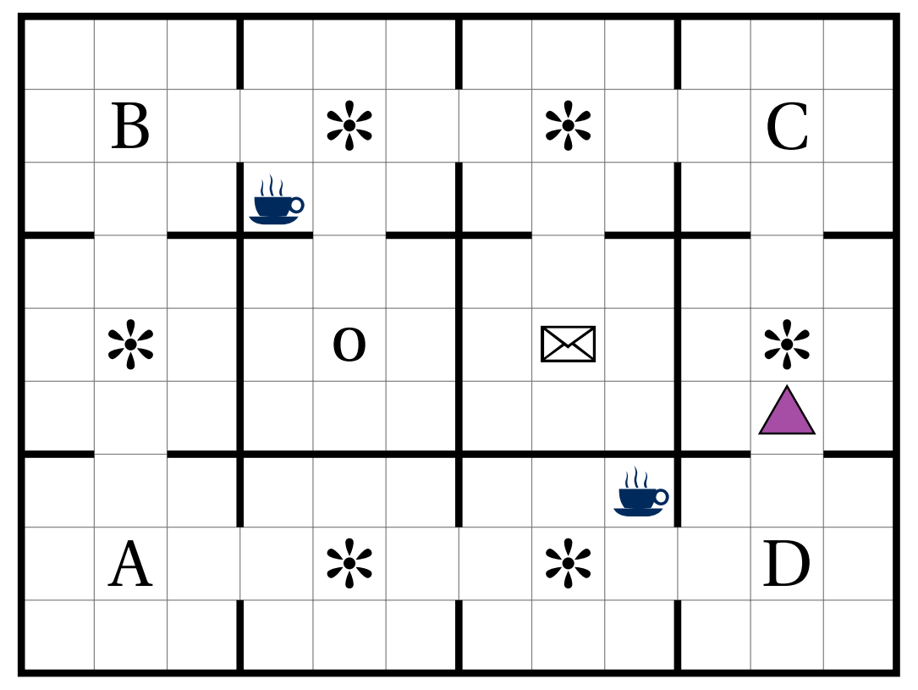
Rodrigo Toro Icarte[†‡] Toryn Q. Klassen[§‡] Richard Valenzano[*] Sheila A. McIlraith[§‡]

[†]Pontificia Universidad Católica de Chile [§]University of Toronto [‡]Vector Institute [*]Ryerson University

[†]rodrigo.toro@ing.puc.cl, [§]{toryn, sheila}@cs.toronto.edu, [*]rick.valenzano@ryerson.ca

Abstract. Reinforcement learning (RL) methods usually treat reward functions as black boxes. As such, these methods must extensively interact with the environment in order to discover rewards and optimal policies. In most RL applications, however, users have to program the reward function and, hence, there is the opportunity to make the reward function visible – to show the reward function's code to the RL agent so it can exploit the function's internal structure to learn optimal policies in a more sample efficient manner. In this paper, we show how to accomplish this idea in two steps. First, we propose reward machines, a type of finite state machine that supports the specification of reward functions while exposing reward function structure. We then describe different methodologies to exploit this structure to support learning, including automated reward shaping, task decomposition, and counterfactual reasoning with off-policy learning. Experiments on tabular and continuous domains, across different tasks and RL agents, show the benefits of exploiting reward structure with respect to sample efficiency and the quality of resultant policies.

Running Example



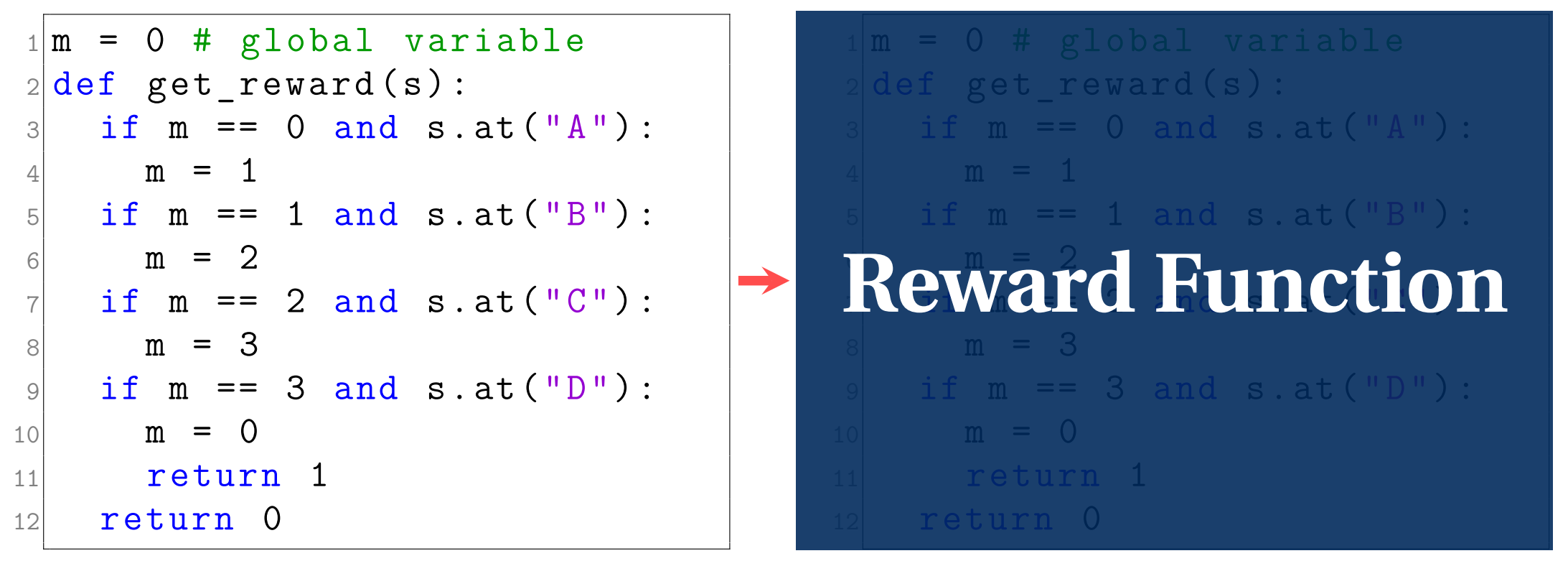
Symbol	Meaning
▲	Agent
✻	Furniture
☕	Coffee machine
✉	Mail room
o	Office
A, B, C, D	Marked locations

Motivation

Task: Patrol A, B, C, and D.

Steps to solve the task using RL:

- Someone programs a reward function.
- The learning agent gets the reward function as a black box.



```
m = 0 # global variable
def get_reward(s):
  if m == 0 and s.at("A"):
    m = 1
  if m == 1 and s.at("B"):
    m = 2
  if m == 2 and s.at("C"):
    m = 3
  if m == 3 and s.at("D"):
    m = 0
    return 1
  return 0
```

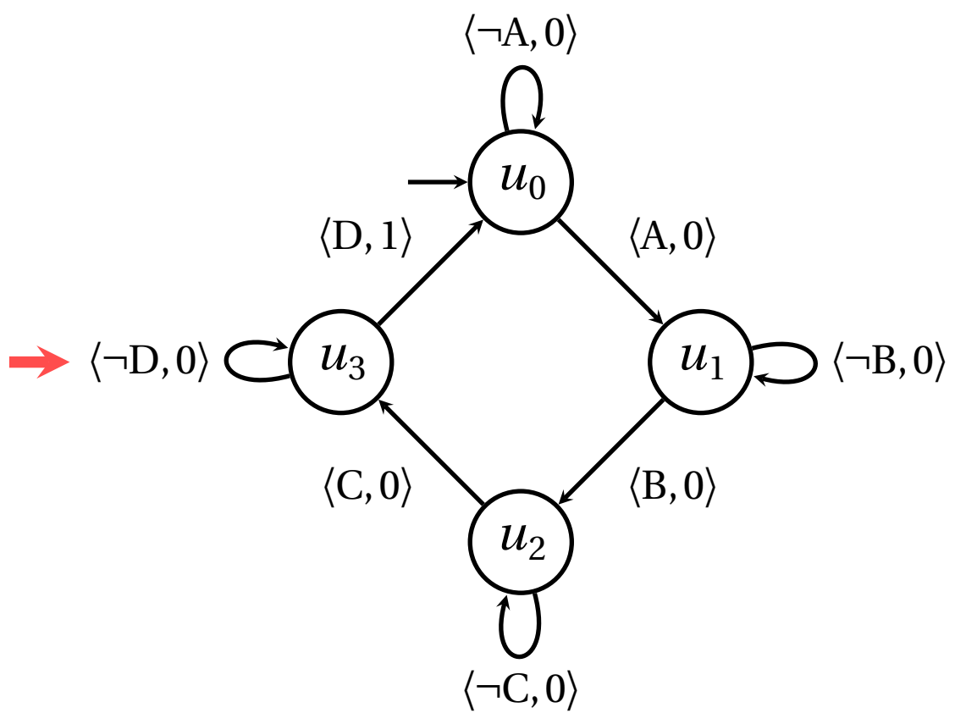
What if we give the agent access to the reward function's code?

Advantage: The agent can exploit the reward structure! How?

What is a Reward Machine (RM)?

Idea: We encode reward functions using a finite state machine.

```
m = 0 # global variable
def get_reward(s):
  if m == 0 and s.at("A"):
    m = 1
  if m == 1 and s.at("B"):
    m = 2
  if m == 2 and s.at("C"):
    m = 3
  if m == 3 and s.at("D"):
    m = 0
    return 1
  return 0
```



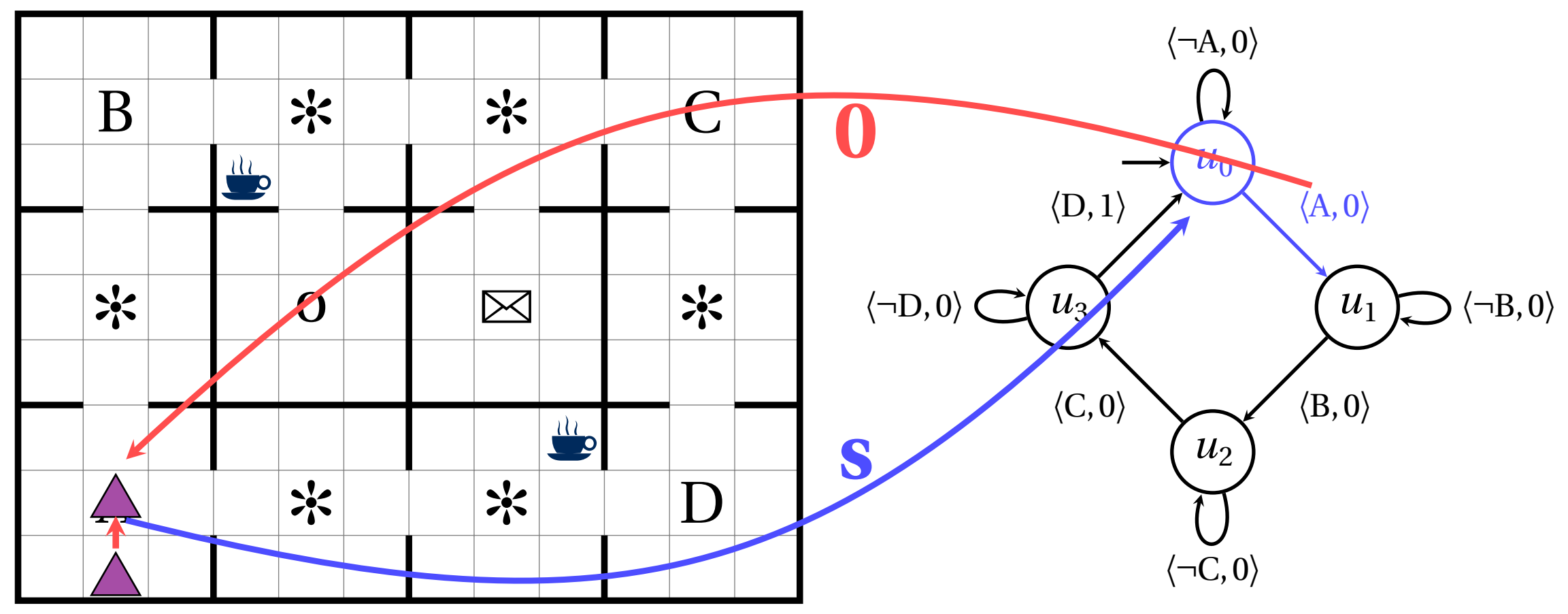
A **reward machine** consists of the following elements:

- A finite set of states U.
- An initial state $u_0 \in U$.
- A set of transitions, each labelled by:
  - a logical condition over properties of the state (e.g. at A, B, ☕, ✉, ...)
  - and a reward function.

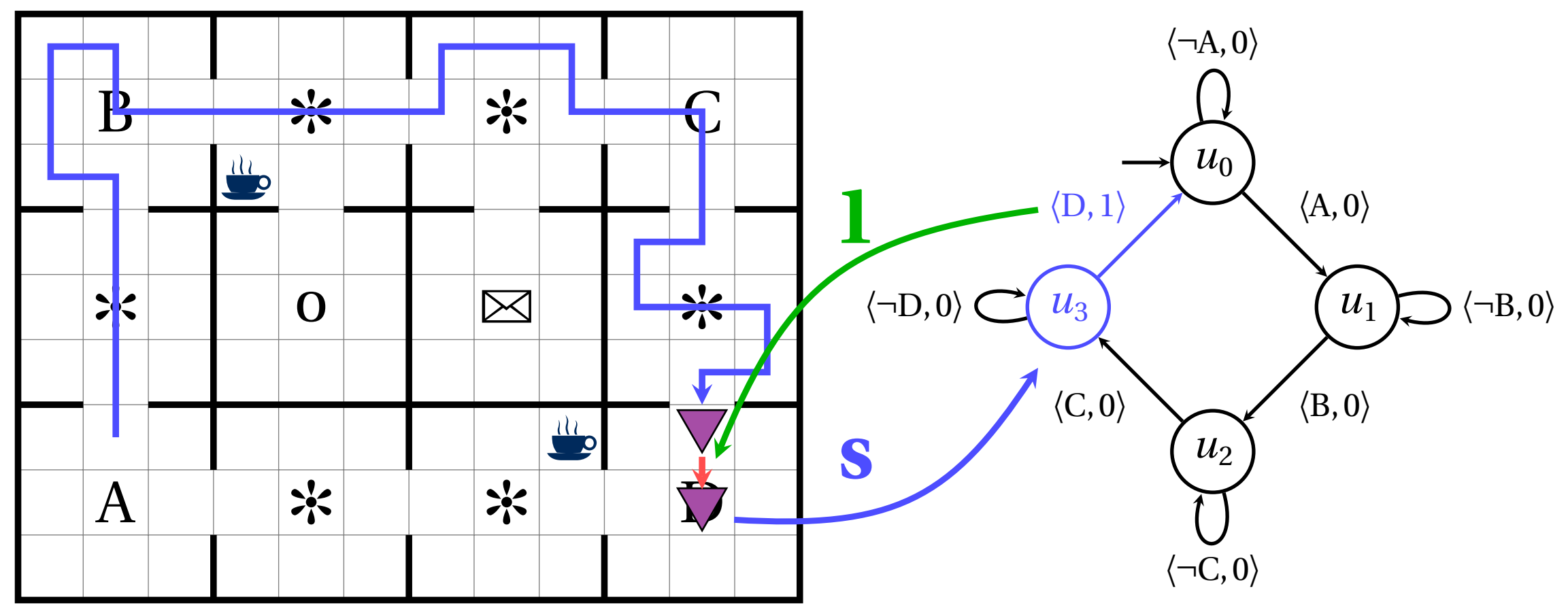
After each move in the environment, an RM makes the transition whose logical condition is satisfied by the environment state, and rewards the agent according to that transition's reward function.

Reward Machines in Action

This RM starts in u_0 and transitions to u_1 when A is reached. The agent gets a reward of 0 from that transition's reward function.



Positive reward is given only when the agent completes a cycle.



Result: $\pi^*(a|s,u)$ patrols A, B, C, and D until the end of time.

How to Learn Optimal Policies Using RMs

The cross-product baseline

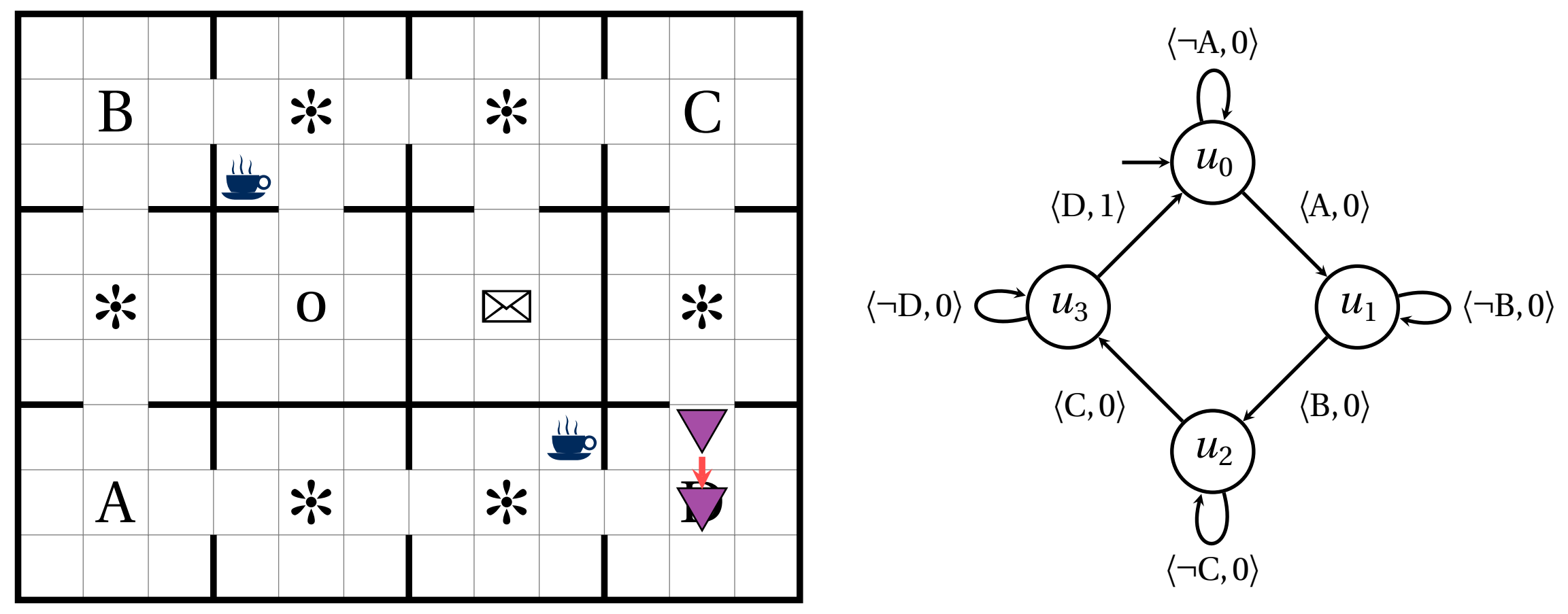
We use any RL method to learn $\pi^*(a|s,u)$

- Observe state $\langle s,u\rangle$ and execute action $a \sim \pi(a|s,u)$.
- Observe next state $\langle s',u'\rangle$ and the reward r.
- Improve policy π using experience $\langle s,u,a,r,s',u'\rangle$.
- $\langle s,u\rangle \leftarrow \langle s',u'\rangle$.

Counterfactual Experiences for RMs (CRM)

We use the RM to generate counterfactual experiences.

- Observe state $\langle s,u\rangle$ and execute action $a \sim \pi(a|s,u)$.
- Observe next state $\langle s',u'\rangle$ and the reward r.
- **Improve policy π using $\langle s,u_i,a,r_{ij},s',u_j\rangle$ for all $u_i \in U$.**
- $\langle s,u\rangle \leftarrow \langle s',u'\rangle$.

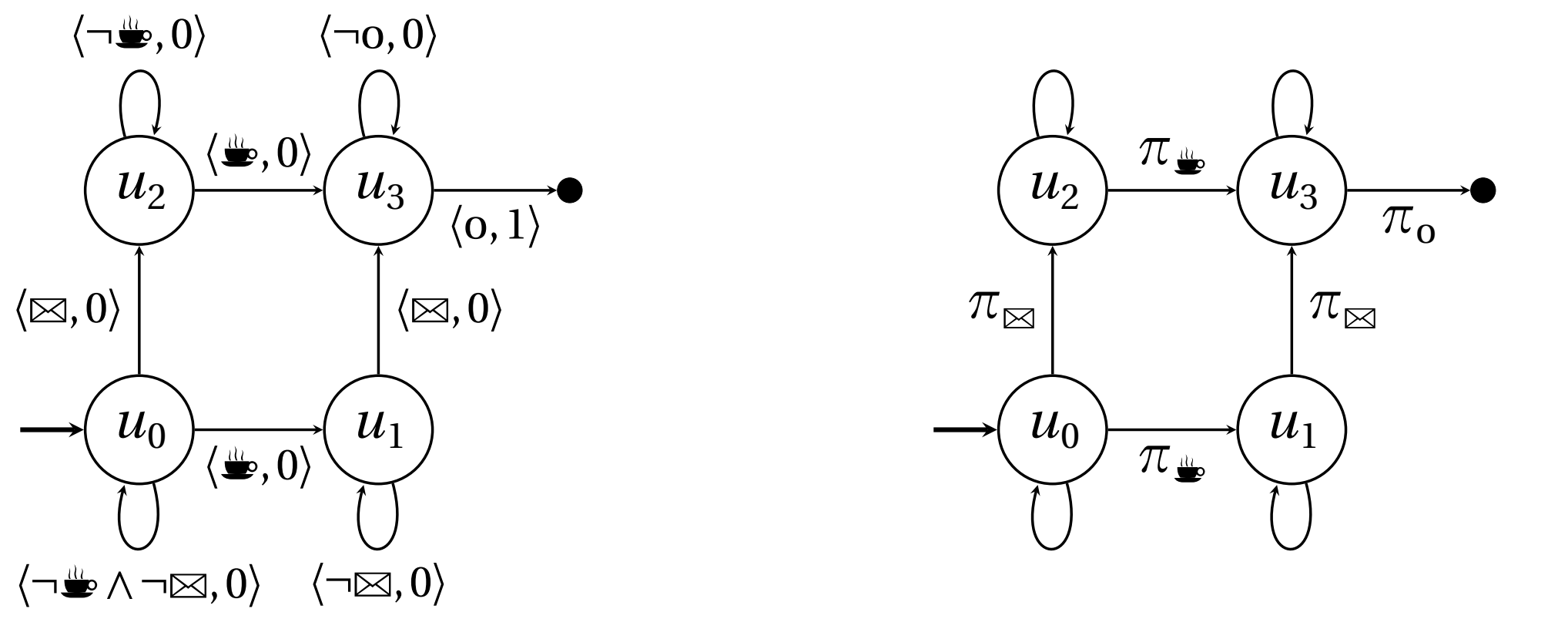


CRM updates	
$\pi \leftarrow \langle s,u_0,\texttt{down},0,s',u_0\rangle$	$\pi \leftarrow \langle s,u_1,\texttt{down},0,s',u_1\rangle$
$\pi \leftarrow \langle s,u_2,\texttt{down},0,s',u_2\rangle$	$\pi \leftarrow \langle s,u_3,\texttt{down},1,s',u_0\rangle$

Theorem: CRM converges to an optimal policy in the limit.

Hierarchical RL for Reward Machines (HRM)

We decompose the overall problem into subproblems that are easier to solve.



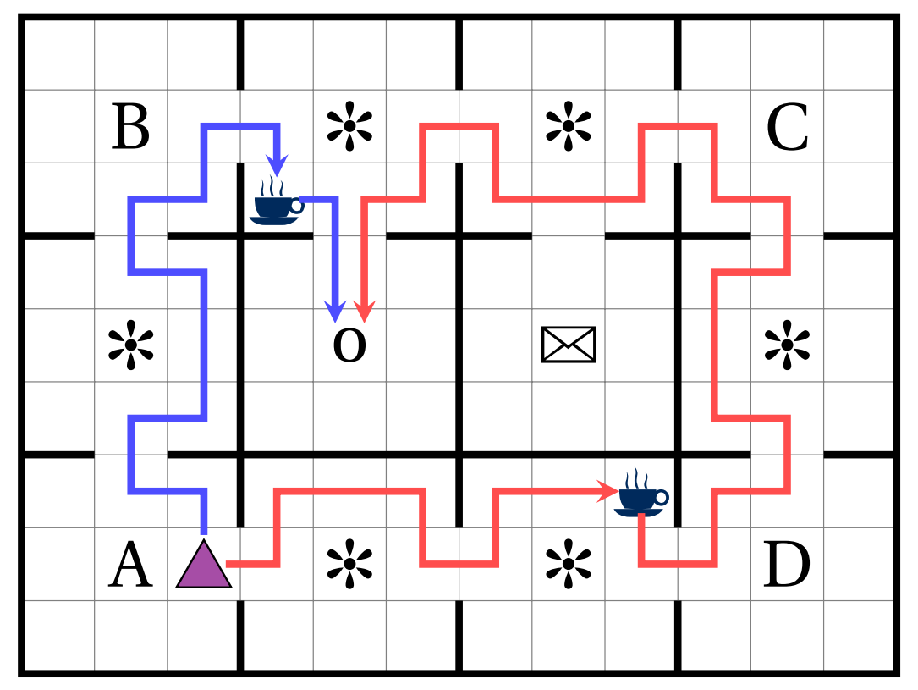
Low-level decisions:

- $\pi_{☕}$: Learns to get ☕as soon as possible.
- $\pi_{✉}$: Learns to get ✉as soon as possible.
- π_{o}: Learns to get to the office as soon as possible.

High-level decisions:

- $\pi(s,u_0) \mapsto \Pr(\{\pi_{✉},\pi_{☕}\})$
- $\pi(s,u_1) \mapsto \Pr(\{\pi_{✉}\})$
- $\pi(s,u_2) \mapsto \Pr(\{\pi_{☕}\})$
- $\pi(s,u_3) \mapsto \Pr(\{\pi_{o}\})$

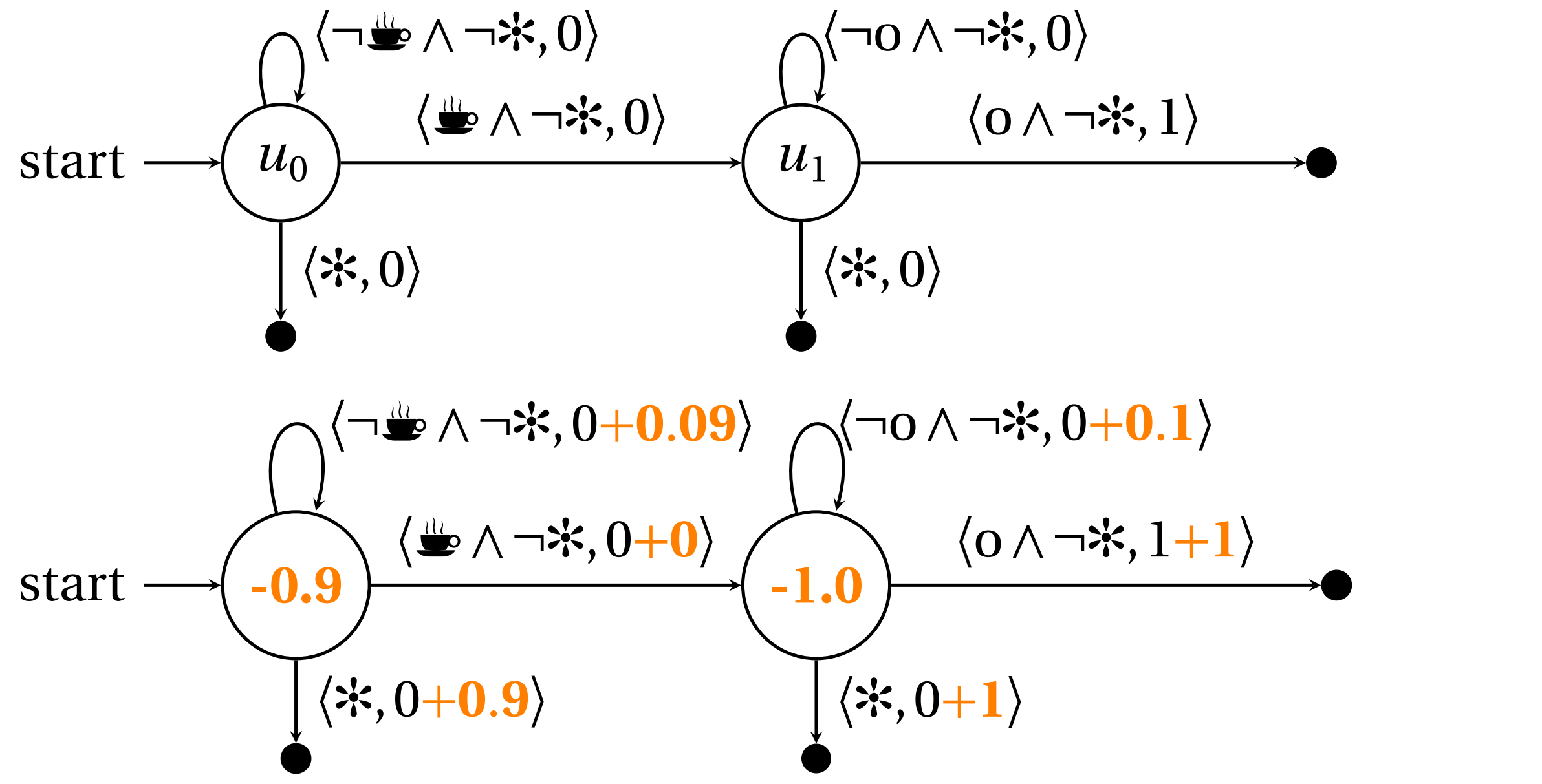
The high-level and low-level policies are learned using RL.



Hierarchical RL might converge to suboptimal policies!

Automated Reward Shaping (RS)

We use the RM to shape the reward function. As a result, we encourage the agent to make progress towards solving the task.

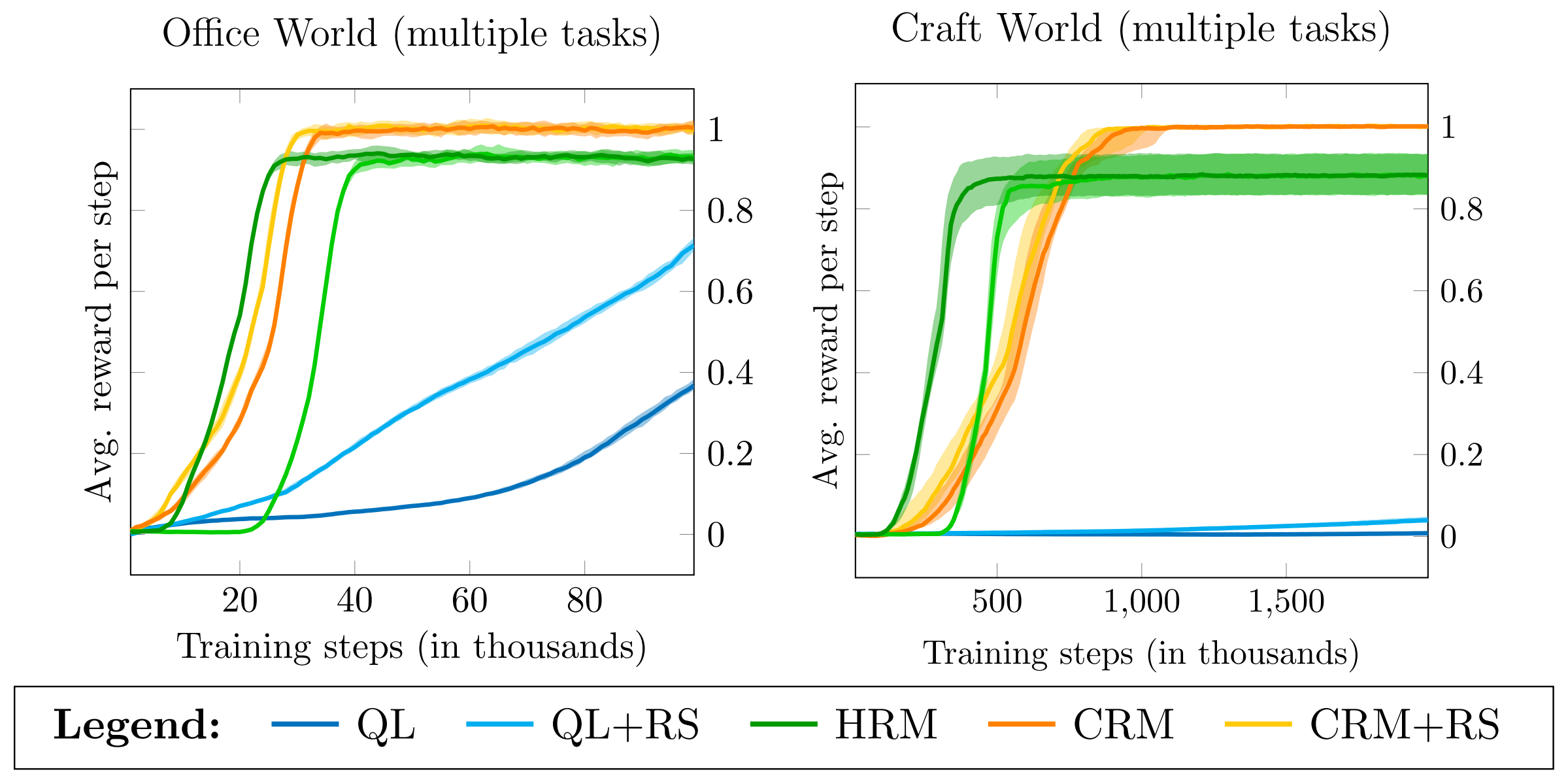


$$r'(u,u') = r(u,u') + \gamma\Phi(u') - \Phi(u) \tag{1}$$

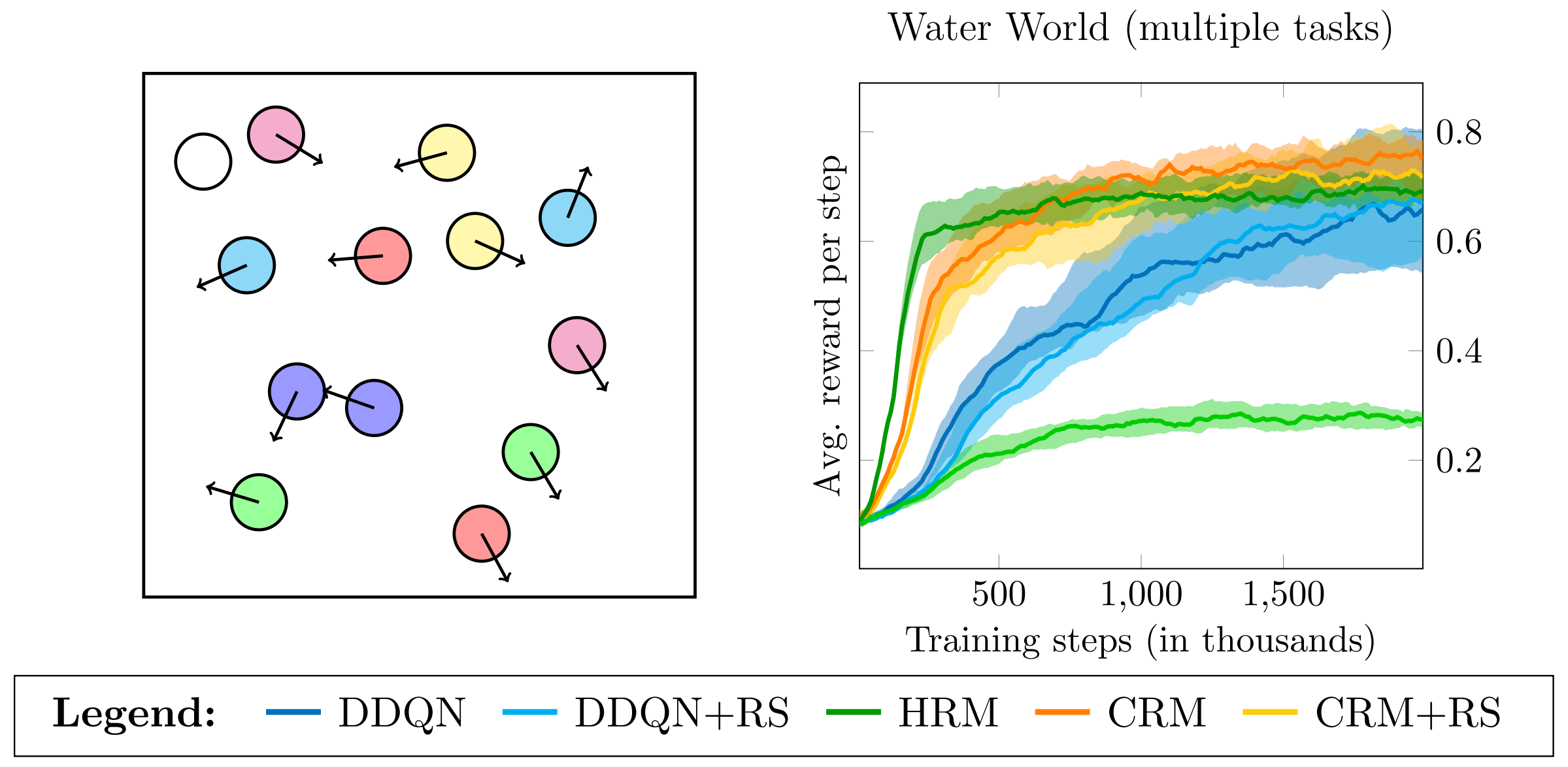
Theorem: RS does not change the set of optimal policies.

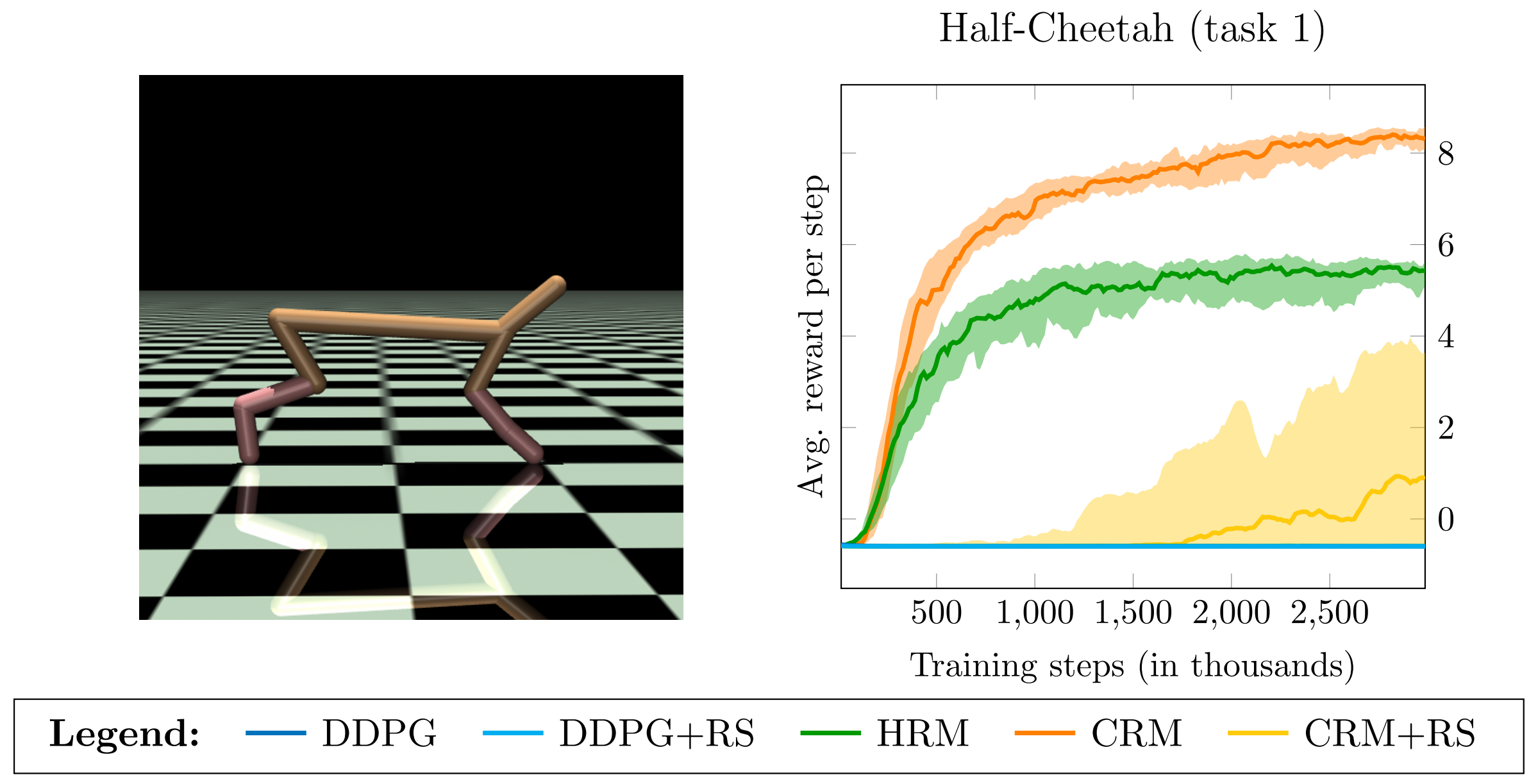
Results

Discrete Domains



Continuous Domains





Discussion

Methods that exploit the structure of the RM (such as CRM and HRM) largely outperform methods that use the RM as a black box (such as QL, DDQN, and DDPG).

Code: `github.com/RodrigoToroIcarte/reward_machines`

Conclusion

> *"To summarize, a nice simple idea exposing more of the structure of an RL problem and the benefits thereof."*
>
> — Third reviewer