DeLF: Designing Learning Environments with Foundation Models

Anonymous submission

Abstract

Reinforcement learning (RL) offers a capable and intuitive structure for the fundamental sequential decision-making problem. Despite impressive breakthroughs, it can still be difficult, especially for non-experts, to use RL in practice in many simple scenarios. In this paper, we provide a formalization for the problem of *environment design*. We propose a method named DeLF: Designing Learning Environments with Foundation Models, that employs Large Language Models to design and codify the user's intended learning environment. In three different learning scenarios, we show that DeLF can obtain executable environment codes for the corresponding RL problems.

Introduction

Reinforcement learning is a powerful paradigm for training intelligent agents through interactions with their environment. Many recent efforts have been dedicated to improving various aspects of RL, such as sample efficiency, reward design, and partial observability. Many environments have also been introduced to showcase the exciting potential of RL. These environments span a spectrum of non-realistic toy examples to realistic safety-critical simulations (Barto, Sutton, and Anderson 1983; Bellemare et al. 2013; Tai et al. 2023). RL or any other decision-making framework will eventually be applied to a learning scenario where all these theoretical and experimental efforts get tested. RL can vastly benefit from the tools that facilitate the work prior to training, such as implementing the learning scenario in a structure that is executable by the RL algorithm. Hence, it's critical to investigate how we design and codify the task and environment setups for the learning scenarios we think of.

Most of the time, researchers and developers go through a repeating cycle of trial-and-error process, playing with parameters and variable definitions in the environment code to finally see some indications of learning. For example, the agent might be able to perform a diverse range of motions and as a result a big action space, but in practice most of the RL algorithms are not able to handle large action spaces. This raises the question of what subset of the action space is a good choice for learning the task. In other situations, we might need to figure out an informative representation of the observation that gives adequate information to the agent for learning the task. As explained, we have a variety of design choices for codifying the environment, but mostly no guaranteed path to find the right design choice. The exhaustive cycle of trial and error reduces the hope of conveniently employing RL in diverse applications. This applies not only to researchers and developers but more importantly to users with less coding and technical skills; considering that experts already have a good intuition on how to define RL components such as observation, action, and reward.

Foundation Models have the potential to fill this gap. The in-context learning abilities of recent language models, together with rich-enough prompts make it possible to generate a RL environment code from user description. Evidently, there is a convenient flow from description to code in large language models specialized for code generation such as GPT-4. In this paper, we investigate the approach of designing and codifying learning environments with coding large language models. We propose DeLF: **De**signing **L**earning Environments with **F**oundation Models and take the first steps toward using this concept in practice. DeLF mainly concentrates on generating executable code from user descriptions. We summarize our contribution as follows.

1. Formalizing the problem of RL Environment Design;
2. Proposing language models as designers of RL environments;
3. Introducing DeLF, a method for designing and codifying RL components with Large Language Models.

We envisage that DeLF can be used in synergy with the recent successful results on designing reward functions with language models (Ma et al. 2023) can make it possible to generate the RL environment code for an arbitrary learning environment; without the user having major coding skills and instead interacting with the underlying foundation model. We believe that DeLF can facilitate codifying environments for researchers and developers who use RL in various applications.

Prelimineries

Foundation Models

Foundation models are usually defined as modeling a (conditional) probability $p(d)$ on a large dataset d (Yang et al. 2023b). In practice, d can be a sequence of text tokens, an image, or a video (sequence of images).

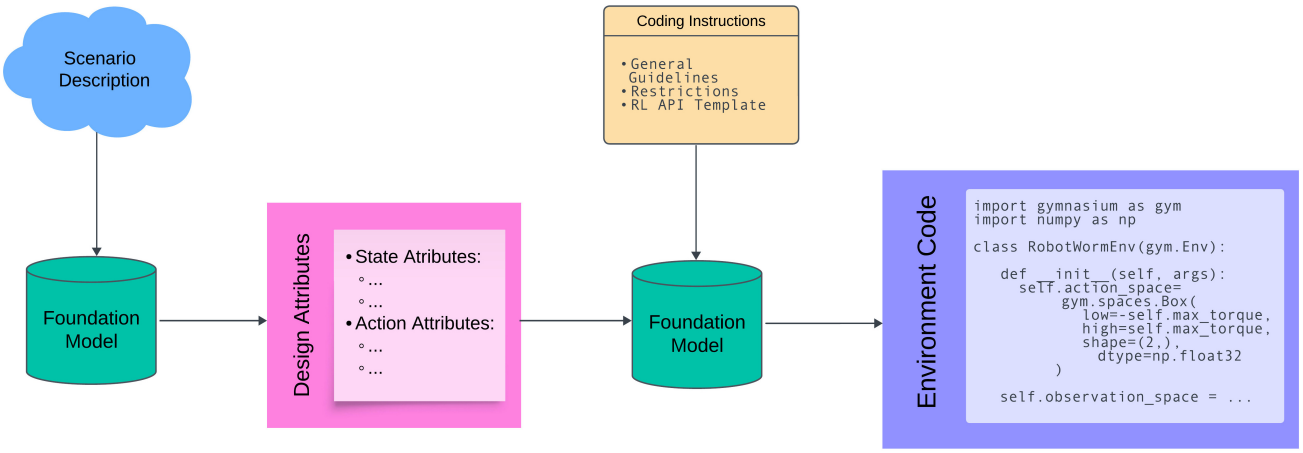

Figure 1: Environment design with DeLF: The user provides a description of a learning scenario to the foundation model (e.g. Large Language Models); the foundation model proposes a design for Observation and Action Attributes; By having the basic template of the user's desirable RL API as context, DeLF is able to generate the environment code which is ready to be fed into a RL Algorithm.

Language Models

Language models are transformer-based sequence models that factorize a joint distribution over a sequence $d = (d_1, d_2, ..., d_l)$ in an auto-regressive manner. (Brown et al. 2020). The models are trained on massive text datasets and are capable of performing a variety of natural language processing tasks.

Reinforcement Learning

Reinforcement learning is an approach to the sequential decision-making problem, where an agent interacts with an environment to perform a specific task. This interaction usually happens in a sequence of discrete timesteps where in each timestep, an agent chooses an action, receives a reward, and transitions to a new state of the environment accordingly. In most of the realistic scenarios, the RL problem is formalized as a Partially Observable Markov Decision Process, POMDP (Kaelbling, Littman, and Cassandra 1998) which is a tuple $\langle S, A, P, R, O, \Omega \rangle$ where S is the state space, A is the action space, $P : S \times A \to \mathbb{P}[S]$ is the transition function, $R : S \times A \to \mathbb{R}$ is the reward function, and O is the agent's observation space, and $\Omega : S \times A \to \mathbb{P}[O]$ is the observation function. We call each member of this tuple, a *component* of the RL Problem, and we try to find the proper design choice for Observation and Action components.

Problem Setting

Definitions

Description Space D is a set of all possible descriptions for a learning environment. A description $d \sim D$ can be in the form of human-language text, an image or a video of the agent performing the task in the environment, a set of mathematical functions. etc.

Component Attributes Att_C is the set of attributes we need to describe the component C.

- **Observation Attributes** Att_O is the set of all the attributes we need to describe the state of the agent in the environment.
- **Action Attributes** Att_A is the set of all the attributes we need to describe the possible actions of the agent in the environment.

Attribute Quantification: For each attribute a, we define a quantification Q_a which is a numerical representation of the attribute. This numerical representation is usually in two forms:

1. Continuous Quantification

$$Q \subseteq [l, u]^n \qquad l, u \in \mathbb{R} \qquad \text{e.g.} \quad [-1, 1]^n \tag{1}$$

2. Discrete Quantification

$$Q \subseteq \mathbb{Z} \qquad \text{e.g.} \quad \{0, 1\} \tag{2}$$

Design Choice of Component C is the tuple $\langle Att_C, Q_C \rangle$ where Att_C is the set of component attributes and Q_C is the set of quantification assigned to each attribute in Att_C.

Task: Task τ is a description of what the agent is supposed to do in the environment. As mentioned before, a description can be in the form of human-language text, an image or a video of the agent performing the task, or a mathematical objective function. etc.

We introduce four notations $\vdash, \nvdash, \Vdash, \nVdash$ for task τ. We write,

$$\langle c_1, c_2, ..., c_n \rangle \vdash \tau \tag{3}$$

if *the design choices of components* $\langle c_1, c_2, ..., c_n \rangle$ *leads to a successful learning of the task* τ. We write,

$$\langle c_1, c_2, ..., c_n \rangle \nvdash \tau \tag{4}$$

if this design choice does not lead to a successful learning of the task. We write,

$$c \Vdash \tau \tag{5}$$

if *given the proper design choices of all other components, the design choice of component* c *leads to successful learning of the task* τ. We write,

$$c \not\Vdash \tau \tag{6}$$

if this choice does not lead to a successful learning of the task.

Here, we are using the term *successful learning* loosely, meaning that the agent has learned a (near) optimal policy in the environment and is acting compatible with the original representation of the task. In practice, a *successful learning* should be determined by analyzing the performance metrics and the agent's behavior on the learned policy.

Sufficient Observation Space: Observation Space O is called *sufficient with respect to task* τ if given all other components designed properly, O leads to a successful learning of the task τ.

$$O \text{ is sufficient } \iff O \Vdash \tau \tag{7}$$

Sufficient Action Space: Action Space A is called *sufficient with respect to task* τ if given all other components designed properly, A leads to the successful learning of the task τ.

$$A \text{ is sufficient } \iff A \Vdash \tau \tag{8}$$

Necessary Observation Space: An observation space O is called *necessary with respect to task* τ if it is the minimal subset of the observation space required for learning the task τ;

$$O \Vdash \tau \quad \text{and} \quad O \setminus \{v\} \not\Vdash \tau \qquad \forall\, v \in O \tag{9}$$

Necessary Action Space: An action space A is called *necessary with respect to task* τ if it is the minimal subset of the action space required for learning the task τ;

$$A \Vdash \tau \quad \text{and} \quad A \setminus \{v\} \not\Vdash \tau \qquad \forall\, v \in A \tag{10}$$

Component Extraction Function $\hat{C} : D \to \langle Att_C, Q_C \rangle$ is a function that maps the description space to the space of design choices for component C. We call this operator a *component designer* who extracts a design choice for component C out of description $d \sim D$.

The Problem of RL Environment Design

We formalize the problem of designing a learning environment as a tuple $\langle \hat{O}, \hat{A}, \hat{R}, I \rangle$ where $\hat{O} : D \to O$ is the observation extraction function, $\hat{A} : D \to A$ is the action extraction function, $\hat{R} : D \to R$ is the action extraction function e, and finally $I : S \times A \to S$ is the agent-environment interaction function.

The function I majorly replicates the agent's transition in the environment and does not need to be extracted. If model-based, I executes the underlying model of the environment. If model-free, we will assume that I has access to the numerical samples of transitions through a simulated or real-world interaction of the agent with the environment.

Language Models as RL Component Designers

Here, we use language models as the extraction function for the attributes, the observation space, and the action space. The input for this extraction function will be the human-language description of the task. The output will be the set of attributes, the codified definition of observation space, and the codified definition of action space, respectively. Practically, we need to provide a context where we explain the general guidelines and templates that we want the language model to follow while generating the outputs. These guidelines can vary based on use-case, and do not restrict the user to follow a certain context template. For clarification, we provide the original prompts that we used as context for our experiments in Appendix B.

Method

DeLF consists of three sections which we refer to as **ICE**; **I**nitiat, **C**ommunication, and **E**valuation. A sufficient design of an environment with a language model is reachable via ICE.

DeLF Initiation

To get the desired learning environment code with fewer prompts, we find it helpful to divide the initial query to the language model into two parts:

- Design: Describe the environment and task to the language model and ask it to extract the observation and action space.
- Codify: Provide the code structure (the intended RL API you want to use to train the agent) as a context for the language model. Optionally, general coding guidelines that you expect the language model to follow.

We found in our experiments that asking for observation and action design choices, in the beginning, will significantly improve the speed and convenience of using DeLFT. This is possibly due to the reason that language models tend to get lost in the middle (Liu et al. 2023), and providing all the description and coding details in one prompt will decrease the significance of the design choice for the language model. This distinction will let the language model focus on a more accurate extraction of observation and action attributes first and then get involved in coding details.

DeLF Communication

For more complex RL Scenarios, the user should expect to communicate with the language model to construct the desired environment code in a few steps. Hence, providing insightful human feedback both after the design query and codify query is essential. After the design query, the user can correct, suggest, and ask the language model about the observation and action attributes. This will help to align the user and language model understanding of the scenario, and avoid obvious misunderstanding and hallucinations of the language model. Also, there might be different styles of writing for describing the environment in the first place; some might provide very detailed info about the environment and some might skip important aspects of the problem.

Communication will help to gradually transfer effective information to the language model and make the design and code aligned with what we want.

DeLF Evaluation

In practice, most RL algorithms ca not handle large action spaces. Besides that, we ideally want to train agents with sufficient and necessary observation. There are no guaranteed procedures to find the proper action and observation attributes before training the agent, and moreover, the agent's performance and alignment heavily rely on other factors such as reward function. In reality, we start with an observation and action that is intuitively compatible with our understanding of the problem and train the agent for a number of timesteps. After seeing the result, we modify the environment code in a loop of trial and error to finally see the intended behavior from the agent. DeLF might be able to reduce the number of unsuccessful trials if provided with reach training feedback from the training evaluation.

Experiments and Results

In this section, we test DeLF for designing three environments with different observation and action characteristics. The example ICE prompts for one of the environments are available in Appendix B.

Grid World

The Grid World environment (Chevalier-Boisvert et al. 2023) is an $n \times n$ surface, with an agent able to move one grid to any of the 4 main directions in each timestep. It's possible to define various scenarios in this environment; here we start with the most basic one, where there is a key and a lock placed in the environment. The agent is supposed to first find the key, and then find and open the lock.

We provide both the prompts and the environment code generated by GPT-4 in Appendix B. In our experiments, despite providing a relatively naive description of the problem, GPT-4 could generate an executable environment code in two trials. Both the action and observation attributes extracted by GPT-4 are compatible with our own understanding of the problem. The two debugging trials were due to minor coding mistakes, such as argument mismatch.

Swimmer

The swimmer environment (Coulom 2002), implemented and popularized as one of the MuJoCo environments (Todorov, Erez, and Tassa 2012), consists of three segments connected to each other by two joints. The agent is able to move by creating torque for each of these joints and the friction caused by the underlying surface.

We use a different name in our description to intentionally reduce the reliance of GPT-4 on the MuJoCo Environment while generating the code. The user input was written based on the basic understanding of one of the authors of the environment. GPT-4 proposed a relatively accurate design choice for observation and action spaces, very close to the expert-designed version of the environment. Besides that, it produced the environment code with less than 10 debugging trials.

Self-Driving Car

We designed a learning scenario for a self-driving agent in a 2-lane street. The agent can accelerate or brake but should stay below a certain speed limit. There are some obstacles placed in the street which the agent must avoid. The task is to reach a certain destination while avoiding obstacles and overspeeding. The closest expert-designed environment to this scenario is HighwayEnv (Leurent 2018).

Despite providing a basic description of the scenario, GPT-4 produced a significantly relevant environment that was ready to execute after a few communication queries. All of the mistakes were considered minor except for one that violated one of the coding rules specified in the codify query.

Discussion and Future Work

Foundation Models

The formalization of the extraction function provided before is not limited to language models. One can imagine that by the time we have capable foundation models for images or videos, we can use them as extraction functions in the problem of environment design. For instance, we can provide the video of the environment and the embodied agent in the input, and ask the foundation model to extract the codified observation and action space.

Metrics

Ideally, we want the design choice of each component to be sufficient and necessary as defined in the problem setting. This is hard to investigate due to a lack of accurate definition for a *successful learning* of the task. While this notion can be problem-specific, a more unified criteria for successful execution of the task can provide reach and accurate feedback to the environment designer.

Reward Design with Language Models

DeLF together with the recent works on using language models for reward design (Yu et al. 2023) can drastically ease the definition and implementation of RL components. In this sense, DeLF is complementary to Eureka (Ma et al. 2023) since the latter gets the environment code as input and generates the reward function for the environment. Hence, it's interesting to see how the synergies between these two work in order to generate the complete implementation of the RL problem based on user description.

Virtual Environment Simulators

DeLF can benefit from virtual simulators to enable the rendering of the environment. With the rendering option comes various usages such as analyzing the agent's behavior throughout and at the end of training. DeLF can utilize the virtual simulators created based on foundation models and enable a more realistic experience for the users. Unisim (Yang et al. 2023a), and RoboGen (Wang et al. 2023) are recent types of simulators that use foundation models to address the challenge of limited data for agentic learning.

Environment Design with DeLF			
Environment	Observation and Action Space	Description Tokens	Trials to Execution
Grid World	Discrete	48	2
Swimmer	Continuous	98	$<$10
Driving Car	Hybrid	135	6

Table 1: Results of DeLF experiments on three learning scenarios. The description token is different from the total number of tokens for each experiment by a fixed number. Trials to execution is the number of extra communication queries needed to reach an executable environment code.

Conclusion

In this paper, we formalized the problem of environment design in RL. First, We introduced the notion of component extraction function that extracts the component's design choice out of user descriptions. Second, We discussed that foundation models can be powerful candidates for the extraction function, due to their abilities to process various user inputs and generate relevant responses. We tested this idea on large language models, by using GPT-4 as the extraction function for observation and action space. Ultimately, We introduced DeLF, a method for designing and codifying RL environments with language models specialized in coding. DeLF showed successful results on three different learning scenarios, generating executable environment codes after a few trials. We tried to take a step forward toward a more practical and broad usage of RL in various applications, and we hope these results create motivation for possible extensions of this approach.

References

Barto, A. G.; Sutton, R. S.; and Anderson, C. W. 1983. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5): 834–846.

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47: 253–279.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901.

Chevalier-Boisvert, M.; Dai, B.; Towers, M.; de Lazcano, R.; Willems, L.; Lahlou, S.; Pal, S.; Castro, P. S.; and Terry, J. 2023. Minigrid & Miniworld: Modular & Customizable Reinforcement Learning Environments for Goal-Oriented Tasks. *CoRR*, abs/2306.13831.

Coulom, R. 2002. *Reinforcement learning using neural networks, with applications to motor control*. Ph.D. thesis, Institut National Polytechnique de Grenoble-INPG.

Kaelbling, L. P.; Littman, M. L.; and Cassandra, A. R. 1998. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1): 99–134.

Leurent, E. 2018. An Environment for Autonomous Driving Decision-Making. https://github.com/eleurent/highway-env.

Liu, N. F.; Lin, K.; Hewitt, J.; Paranjape, A.; Bevilacqua, M.; Petroni, F.; and Liang, P. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*.

Ma, Y. J.; Liang, W.; Wang, G.; Huang, D.-A.; Bastani, O.; Jayaraman, D.; Zhu, Y.; Fan, L.; and Anandkumar, A. 2023. Eureka: Human-Level Reward Design via Coding Large Language Models. *arXiv preprint arXiv:2310.12931*.

Tai, J. J.; Wong, J.; Innocente, M.; Horri, N.; Brusey, J.; and Phang, S. K. 2023. PyFlyt–UAV Simulation Environments for Reinforcement Learning Research. *arXiv preprint arXiv:2304.01305*.

Todorov, E.; Erez, T.; and Tassa, Y. 2012. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, 5026–5033. IEEE.

Wang, Y.; Xian, Z.; Chen, F.; Wang, T.-H.; Wang, Y.; Fragkiadaki, K.; Erickson, Z.; Held, D.; and Gan, C. 2023. RoboGen: Towards Unleashing Infinite Data for Automated Robot Learning via Generative Simulation. *arXiv preprint arXiv:2311.01455*.

Yang, M.; Du, Y.; Ghasemipour, K.; Tompson, J.; Schuurmans, D.; and Abbeel, P. 2023a. Learning Interactive Real-World Simulators. *arXiv preprint arXiv:2310.06114*.

Yang, S.; Nachum, O.; Du, Y.; Wei, J.; Abbeel, P.; and Schuurmans, D. 2023b. Foundation models for decision making: Problems, methods, and opportunities. *arXiv preprint arXiv:2303.04129*.

Yu, W.; Gileadi, N.; Fu, C.; Kirmani, S.; Lee, K.-H.; Arenas, M. G.; Chiang, H.-T. L.; Erez, T.; Hasenclever, L.; Humplik, J.; et al. 2023. Language to Rewards for Robotic Skill Synthesis. *arXiv preprint arXiv:2306.08647*.

Appendix

A. Codes

The link to GitHub repo containing GPT-generated codes will be available here. (Not hyper-refed now, due to anonymity). We provided the code for one of the experiments in Appendix C. in the

B. Design and Codify queries

Here are the original prompts we used for our experiments. We followed the same template as shown below for the all of our experiments.

B.1. Design Query

The design prompt is basically the description of the problem provided by the user; and querying the user to extract the design attributes.

```
I want to train a Reinforcement Learning Agent for a learning scenario, and I want you to design the RL environment components of this scenario.

Here is the scenario:
<Describe the scenario here>

What's the observation space and action space for this scenario? List the attributes of observation and action and justify why these sets of attributes are enough for the agent to learn the task in this environment.
```

B.2. Codify Query

The codify query contains general coding instructions and the environment code format from OpenAI gym API. The code format is basically for enforcing the input-output format of major functions. This make the environment code directly executable with commonly used RL algorithms. You can add your own coding rules and desire RL API.

```
I want you to codify the environment for the scenario I described before. I want you to follow the format of the OpenAI gymnasium environments library provided below. It's recommended to start with "import gymnasium as gym".
Fill out the functions and generate code where specified so that we can train this agent. You can add new functions if you want but Don't change input-output format of the existing functions.
You should initialize self.observation_space and self.action_space in the init function. Use the predefined spaces from gymnasium.spaces to initialize them.
You are free to use Python libraries such as numpy in the environment. Just remember to import them.


class EnvName(gym.Env):

    def __init__(self, args):

        # Generate Code Here



    def _perform_transition(self, action):

        # Generate Code Here

        return  observation



    def _get_reward(self, action, observation, terminated):

      # Generate Code Here

      return r


    def _is_terminated(self):

      # Generate Code Here

      return terminated


  def _get_info(self):

    # Generate Code Here If Necessary

      return info


  def step(self, action):

      observation = self._perform_transition(action)
      terminated = self_is_terminated(observation)
      reward = self._get_reward(action, observation, terminated)
      info = self._get_info()

      return observation, reward, terminated, False, info







    def reset(self, seed=None, options=None):
      super().reset(seed=seed)

    # Generate Code Here

      return observation, info
```

C. Experiments

C.1. GPT-generated code for Gridworld Environment

```python
import gymnasium as gym
import numpy as np
from gym.spaces import Discrete, Box

class KeyLockEnv(gym.Env):
    metadata = {'render.modes': ['human']}

    def __init__(self, grid_size=8):
        super(KeyLockEnv, self).__init__()

        # Grid size
        self.grid_size = grid_size

        # Observation space: Agent's pos (x, y), Key's pos (x, y), Lock's pos (x, y), Key status, Lock status
        self.observation_space = Box(low=0, high=grid_size, shape=(8,), dtype=np.int32)

        # Action space: 0 = Move North, 1 = Move South, 2 = Move East, 3 = Move West, 4 = Pick Up Key, 5 = Open Lock
        self.action_space = Discrete(6)

        # Initialize state
        self.state = None
        self.reset()

    def _perform_transition(self, action):
        # Update the agent's position based on the action
        if action == 0 and self.state[1] > 0:  # Move North
            self.state[1] -= 1
        elif action == 1 and self.state[1] < self.grid_size - 1:  # Move South
            self.state[1] += 1
        elif action == 2 and self.state[0] < self.grid_size - 1:  # Move East
            self.state[0] += 1
        elif action == 3 and self.state[0] > 0:  # Move West
            self.state[0] -= 1
        elif action == 4:  # Pick Up Key
            if self.state[0] == self.state[2] and self.state[1] == self.state[3]:
                self.state[6] = 1  # Key is picked up
        elif action == 5:  # Open Lock
            if self.state[0] == self.state[4] and self.state[1] == self.state[5] and self.state[6] == 1:
                self.state[7] = 1  # Lock is opened

        return self.state

    def _get_reward(self, action, observation, terminated):
        # Simple reward function
        if terminated:
            return 100  # High reward for completing the task
        elif action == 4 and observation[6] == 1:
            return 10  # Reward for picking up the key
        elif action == 5 and observation[7] == 1:
            return 50  # Reward for opening the lock
        return -1  # Small penalty for each step

    def _is_terminated(self):
        # The episode terminates when the lock is opened
        return self.state[7] == 1

    def _get_info(self):
        # Additional info about the environment
        return {}

    def step(self, action):
        observation = self._perform_transition(action)
        terminated = self._is_terminated()
        reward = self._get_reward(action, observation, terminated)
        info = self._get_info()

        return observation, reward, terminated, info

    def reset(self, seed=None, options=None):
        super().reset(seed=seed)

        # Randomly place the agent, key, and lock in the grid
        positions = np.random.choice(self.grid_size * self.grid_size, 3, replace=False)
        self.state = np.array([positions[0] % self.grid_size, positions[0] // self.grid_size,  # Agent's position
                               positions[1] % self.grid_size, positions[1] // self.grid_size,  # Key's position
                               positions[2] % self.grid_size, positions[2] // self.grid_size,  # Lock's position
                               0,  # Key's status (not picked up)
```

```
78                                  0])  #
    Lock's status (not opened)
79
80          return self.state, self.
    _get_info()
```