Investigate LLM-based World Models for Environment-Descriptive Language Understanding Tasks

Joe Nguyen   Andrew Le   Alexander Mote
Oregon State University
{nguyejoe, leandr, motea}@oregonstate.edu

Abstract

To interact effectively with humans in the real world, it is important for agents to understand language that describes the dynamics of the environment—that is, how the environment behaves—rather than just task instructions specifying what to do. For example, a janitorial robot understanding a dynamics-descriptive statement like "there is usually trash left in the common room after the weekend" provides contextual information about environmental changes, beyond direct instructions like "pick up trash in the common room". Recent works have addressed this problem using a model-based approach, where dynamics-descriptive language is incorporated into a world model which can predict the outcome of agent actions according to the described dynamics. An behavior policy is then derived from this language-conditioned world model. However, these existing methods train the world model by supervised learning, which takes time in collecting data and actual training. Recent works such as CWM (1) and WorldCoder (7) try to solve this problem by using a Large Language Model (LLM) to synthesize a world model in the format of code with limited training data, therefore bypassing intensive training costs. However, environments they use do not require agents to understand human language, especially environment-descriptive language, nor language grounding ability. To fill this gap, we take CWM as the baseline, and investigate whether it can build a language-conditioned world model that requires both environment-descriptive language understanding and grounding. We use MESSENGER (stage S1) (4) as the testbed environment and find that CWM can produce a perfect world model, only with a well-parsed language form and given language grounding. However, even S1 is the simplest setting of MESSENGER, CWM fails to produce a world model code that can 1) understand language in human natural form and 2) infer language grounding from training dataset. We argue that these findings are crucial for us to improve current LLM-based model-based systems for language understanding tasks, especially those with environment-descriptive language.

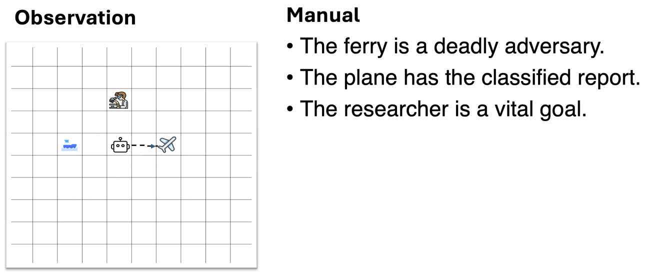

Observation
Manual
• The ferry is a deadly adversary.
• The plane has the classified report.
• The researcher is a vital goal.

Figure 1. An example game of MESSENGER S1. In this game, the entity does not have message at the beginning of the game. Therefore, it goes to the messenger to retrieve the message and ends the game. All entities except the agent are stationary, thus the manual only describes roles associated with entity names.

1. Introduction

We envision the future where humans can communicate with AI agents to automate repetitive tasks in the real world. Traditionally, language is used for specific task instructions, i.e. *what to do*. For example, "go to the door" or "clean the house". However, language can also provides information about environments. Instead of receiving task instruction like "pick up trash in the living room", a robot might receive: "there is usually trash left in the living room over the weekend". Such environmental description not only makes human interaction more natural and intuitive, but also provides important contextual information about how the environment changes over time. It informs the agent about *how the environment behaves*—its dynamics, the current state of the world, and how various entities interact with each other and with the agent—not just *what to do*. Therefore, it is important to make AI agents understand such language, enabling them to make better decisions, anticipating potential hazards and adapt effectively to environmental changes.

We illustrate this problem by using a simple 2D grid-based game, shown in Figure 1. Each game instance consists of several entities, an agent positioned in a 10×10 grid-world observation (on the left), and a language manual (on the right). The agent (depicted as 🤖) acts as a courier, tasked with picking up a message *or* delivering it to a goal while avoiding an enemy. The language manual provides descriptions of the entity attributes, helping the agent understand the environment's dynamics: what the roles of entities are and how the environment changes as the agent interacts with them. For example, the sentence "The ferry is a deadly adversary" indicates that the ferry is an enemy and will cause the agent to fail the task if it touches the ferry. To succeed, the agent must interpret the language manual, identify the entities, and infer their respective roles based on observed behaviors rather than explicit labels[1].

Language is particularly valuable because it allows for the description of new scenarios by recombining known terms. For instance, the following example manual describes a novel set of dynamics derived from known concepts in the manual shown in Figure 1. To succeed in this environment—where the dynamics have changed but the rule remains the same—the agent must adopt a different behavior than in the scenario in Figure 1.

> The ferry is the goal you need to go to.
> The stationary plane is an enemy.
> The researcher is an important message.

Our goal is to develop an agent capable of understanding dynamics-descriptive language by grounding it to entities. More importantly, we aim for the agent to generalize to unseen dynamics described by novel language, allowing it to adapt its behavior to new environment changes. In the current literature, there are two main approaches to building such an agent: (1) using model-free reinforcement learning, or (2) a model-based approach where we train a language-conditioned world model from which a policy is derived. However, both approaches typically assume access to large amounts of experience gathered through environment interaction or expert demonstrations—an assumption that may not hold in many real-world settings.

We aim to explore how agents can understand dynamics-descriptive language and act appropriately with the *minimal* amount of training experience. We hypothesize that this can be achieved by leveraging Large Language Model (LLM), which possess extensive prior world knowledge accessible through a "language API." We focus on model-based approach as we are interested in how good LLM can represent the language-conditioned world model and how we can use this world model to further improve a trained behavior policy.

[1]The readers are encouraged to go through the example, infer the roles of entities in the observation, and derive an agent behavior to understand how the agent reads the language to complete the task.

In this project, we explore whether LLMs can construct an accurate world model of the environment from environment-descriptive language, enabling effective policy derivation without requiring substantial real-world interaction / demonstration. To answer these questions, we adopt the most recent works on LLM-based world models: WorldCoder (7), CWM (1), and PoE-World (6). We choose CWM as the baseline and the environment MESSENGER (4) with Stage S1 (S1) as the testbed environment, where the agent needs to understand environment descriptions to solve the tasks. The results show that only with given language grounding embedd to the prompt for LLM, CWM can produce a perfect world model code. However, when it is asked to find the language grounding based on traning dataset, CWM fails to produce a world model code that understands human language in natural form and can solve language grounding, even the simplest setting S1 of MESSENGER.

2. Problem Setting

We define our problem as a Language-conditioned Markov Decision Process, represented by the tuple $(\mathcal{S}, \mathcal{A}, r, T, L, \gamma, H, L, \mathcal{E})$, where:

- $\mathcal{S}$ is the state space where each $s \in \mathcal{S}$ is a grid-world observation containing a subset of N distinct entities from entities $\mathcal{E}$ and the agent,
- $\mathcal{A} = \{\texttt{up, down, right, left, stay}\}$ is the discrete agent action space,
- L is a language manual featuring transition function T and reward function r. L consists of N sentences associated with N entities, where each sentence describes one entity, as shown in Figure 1.
- $r(s, a)$ is the reward function for a given state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, described by the language manual L,
- $T(s'|s, a)$ is the transition function, defining the probability distribution of the next state s' given the current state s, action a. The transition function is described by the language manual L.
- H is the decision horizon,
- $\gamma \in [0, 1)$ is the discount factor.

In our problem, the agent must take a sequence of actions $a_t \in \mathcal{A}$, where time step $t \in [1..H]$, resulting in a state-action trajectory $(s_1, a_1, \ldots, s_H, a_H)$. The agent needs to find a policy $\pi : \mathcal{S} \times L \to \mathcal{A}$ that maximizes the expected sum of discounted rewards:

$$\mathbb{E}_{\pi, L}\left[\sum_{t=1}^{H} \gamma^{t-1} r(s_t, a_t)\right] \tag{1}$$

To test the policy understanding and generalization of language, we want to have a set of language manuals $\mathcal{L}$ into three subsets $\mathcal{L}_{train}, \mathcal{L}_{dev}, \mathcal{L}_{test}$, where $\mathcal{L}_{dev}$ and $\mathcal{L}_{test}$ are new languages describing novel combinations of known concepts in $\mathcal{L}_{train}$.

2.1. Environment: MESSENGER

To study how the policy can generalize over new language describing new dynamics of the environments, we utilize the MESSENGER environment (4). As shown in Figure 1, MESSENGER is a 10×10 grid-world environment. Each game includes an observation (displayed on the left in the figure) and a language manual (shown on the right) containing various entities and a single agent (or player). Each entity is assigned one of three possible roles: `enemy`, `messenger`, or `goal`. The objective is to first reach the messenger to get the message and then reach the goal while avoiding enemies. The roles are unknown from the observation but available to the agent from the language manual. The agent then must read the manual to ground entities to their associated roles to complete the task. MESSENGER provides three stages with different levels of language generalization assessment; these stages are discussed in more detail in Appendix 6.1.

Manuals. There are twelve different entities (e.g., `airplane`, `researcher`, etc.) denoted by a fixed set of corresponding symbols that are used consistently across game instances, e.g. symbol ✈ is used across games that involve entity `airplane` in the observation shown in Figure 1. Note that the observation does not have entity names (e.g. `airplane`), requiring the agent to observe the entity's symbol and ground the entity's name to its corresponding symbols from the manual.

There are also three movement types for entities: `moving`, `fleeing`, and `stationary`, which describe movement trends relative to the agent's position (e.g., the manual describes "heading closer and closer to where you are" for the movement type `moving`).

For each game, the game engine assigns different roles (`enemy, goal, messenger`) and movement types (`moving, fleeing, stationary`) to a set of entities, along with the associated language manuals containing this information, e.g. "the plane fleeing from you has the classified report". As a result, two games with the same set of entities and identical grid-world observations can have different language manuals and, consequently, different reward and transition functions.

Each entity is assigned a movement type and a role along with its name, all of which are unavailable in the observation but described in the language manual. To succeed in the game, the agent must understand the language manuals L to ground the roles and movement types to the entities. This understanding is essential for inferring the underlying transition function T, the reward function $r(s, a)$, and the termination condition. Specifically, the agent must read the manual, observe entity symbols and their behaviors to infer entity names and movement types, and then ground language sentences to each entity. The agent then grounds the roles described in the manual to each entity. For example, given the game in Figure 1, the agent must first ground the term `plane` to the symbol ✈ thus finding the correct sentence in the manual describing it: "The plane has the classified report.". This sentence describes the entity-movement-role assignment: `plane` - `NA` - `messenger`. The agent then is able to find the role of this entity is `messenger`, thus going towards it to get the message.

Environment Dynamics and Action. In stage S1, the agent wins the game by completing the following mission, depending on its initial state:

- if initially the agent does not have the message: it needs to go to the message.
- if initially the agent already have the message: it needs to go to the goal.

If the agent can win the game, the environment returns a reward of 1. The agent loses the game and get the reward of -1 if any of following event occurs:

- The agent fails to complete the mission before the time limit ends (which is four steps in S1) .
- The agent hits the enemy.
- The agent hits the wrong entity, e.g. the agent with the message hits the messenger or the agent without the message hits the goal.

In stage S2 and S3, the agent loses the game and incurs -1 reward[2] if either of two events occurs - the agent is in the same cell as the enemy or reaches the goal without first getting the message. Reaching the messenger gives the agent a reward of 0.5, and then reaching the goal provides a reward of 1.

The agent can navigate the grid using five actions: `left`, `right`, `up`, `down`, and `stay`. The agent can only interact with entities when it is in the same grid cell as the entity[3].

[2]In the $S3$ setting of the game, there is a inconsistency in the environment implementation with the description provided in (4): when the agent collides with two enemy entities, the environment returns a reward of -2 instead of -1. We observe that this rarely happens and thus have no significant impact on expected policy's behavior.

[3]We observed a inconsistency in the implementation with the environment description outlined in (4). The agent can collide with entities even when they are not in the same grid cell. This is however deemed acceptable to the policy as the agent is still able to try to either go back or stay away from the other entity. More details can be found in this discussion: https://github.com/ahjwang/messenger-emma/issues/6

3. Experiments

We want to investigate two research questions: 1) Can LLM-based methods produce good Python-based world models from *language grounding* given observation and language descriptions (section 3.1), and 2) Given a Python-based world model, can we derive a good policy (section 3.2)? To answer these, we adopt CWM (1) as the current Python-based world model generated from LLM. [4] We use Qwen32B-Coder (5) as our LLM base models.

Environment setup. We adopt MESSENEGER Stage 1 (S1) (4) as our testbed environment. Given CWM uses only text-based observation, we convert the observation of S1 from a grid-based representation to a discrete one, which consists of a dictionary of the following values:

- entity ids
- entity positions
- avatar id
- avatar position

Since Python-based format can not understand human language in the natural form, we simplify the manuals by parsing them into a predefined format. Specifically, each sentence is parsed into a dictionary of "name" and "role", e.g. {`name: airplane, role: messenger`}.

3.1. World model evaluation

Set up. We want to investigate whether CWM can perform language grounding (entity ids to entity name mapping, e.g "bird is entity id 4") and produce a good language-conditioned world model. To answer this, we run two experiments:

- LLM with given language grounding (LLM w/ LG): we provide the grounding in the prompt for LLM. See prompts we use in details in Appendix 6.2.
- LLM without given language grounding (LLM w/o LG): in this case, LLM has to learn continually language grounding from training dataset. Since CWM does not have an mechanism to build a Python world model code that *is able to learn* from training dataset, it is *impossible* to immediately adapt CWM to infer language grounding. We therefore, propose a simple change in LLM prompts that ask LLM to create a program can learn language grounding from the observation, save this to disk. We then reload this language grounding and refine it whenever we call the action "Improve" the code (meaning we prompt LLM to improve current code with one training sample that is predicted incorrect by the current code). For more details, please refer to Appendix 6.3.

Given CWM uses only 5 random trajectories and further 5 sub-optimal demonstrations to prompt LLM, we just use 10 random training environment steps total. We collect 10000 environment steps in test environments using optimal, sub-optimal, and random policies to make sure test state space coverages enough cases. [5] We report next state accuracy, termination precision/recall, reward precision/recall as our evaluation metrics in Table 1.

We can observe that when world model requires language grounding (CWM w/t LG), the world model degrades significantly. This indicates we need to have a more advanced method to make LLM produce a code that can learn language grounding from training data.

We show world model code for CWM w/LG in Appendix 6.5 and w/t LG in Appendix 6.4.

3.2. Policy evaluation

Due to time constraint, we only evaluate policy derived from CWM w/ LG world model in test environment, using the same procedure with MCTS used in CWM. Please refer the original paper for detailed planning procedure and hyperparameters. The policy reward increased to 0.56, which indicates that the agent is reaching the correct entity in roughly 78% of all episodes. We have not finetune hyperparameters for MCTS but we suspect the policy derived from CWM w/LG should be perfect given the world model with language grounding is perfect in Table 1.

3.3. Discussion

We want to highlight several insights. World model code generated by CWM

Fails to understand natural language. CWM assumes that the world model can be represented by Python code. However, if the world model needs to understand human language, Python code is unable to understand human language unless it calls external API to parse the language. This can be a future work where we include API calls into Python world model code.

[4]Though we wished to adapt both CWM and WorldCoder to MESSENGER, we were not able to fully adapt WorldCoder. In particular, we were able to adapt the initialization stage of the WorldCoder algorithm. That is, we were able to generate the initial prompt for the LLM to return a valid initial transition and reward function. However, we were not able to properly implement WorldCoder's program refinement process. The program refinement process requires collecting observations from the world model and evaluating them against observations from the environment. We were not able to correctly collect observations using the MESSENGER environment.

[5]Here is the counter of rewards in test steps: Counter(0: 6729, -1: 2683, 1: 588)

Method	State Accuracy	Termination recall	Termination precision	Reward recall	Reward precision
CWM w/ LG	0.99 ± 0.01	1 ± 0	1 ± 0	1 ± 0	1 ± 0
CWM w/t LG	1	1	0	0.58 ± 0.05	0.53 ± 0.1

Table 1. World model evaluation. We report mean and std for each cell, where we run the experiments three different seeds, unless where we do not have symbol $\pm$, we only run 1 seed due to the time constraint.

Fails to learn language grounding from training data. The world model code can learn from data only in the action "Improve" from CWM. However, at each node of the tree which CWM expands, CWM lacks a mechanism to load already learnt entity id to entity name mapping from the previous node (the parent node of the current node). CWM also does not have a specialized prompt to force LLM to generate reasoning tokens that think to update the current entity id to entity name mapping given the incorrect predictions.

CWM only works well when environment information is completely correct and super detailed. Human is not perfect and make mistakes, yet CWM does not have a mechanism to ask clarification questions and seek for more details. Including human in the loop might speed up prompt writing and overal process.

4. Related Work

LLM for world model. There are two ways to use LLM for world modeling. First, LLM can be used as a *direct simulator* (8) for the agent. However, the LLM still struggles to model transition functions that require commonsense reasoning or domain knowledge. Therefore, the second approach, which is more common, is to use LLM to *build an abstract representation of the world*, from which a policy can be derived. Recent works choose programming languages as the promising choice for world modeling, thanks to its fast inference and interpretable behavior. One such programming language choice is Planning Domain Definition Language (PDDL), which expresses a list of possible actions (e.g. `put <object> on a table`) and their preconditions (e.g. `exists table AND exists <object>`) and effects on the environment (e.g. `<objects> on the table`). However, PDDL approaches in (3) (10) require either human annotation of action descriptions or human feedback to refine PDDL codes. Another option is to use a probabilistic programming language called CHURCH (2), which is encoded by a LLM given text-based observations (9).

A more recent approach is to input multi-modal environment observations to LLM and then generate a Python-based world model, represented by WorldCoder (7), CWM (1) and PoE-World (6). WorldCoder maintains a set of programs which have some degree of success, and uses an LLM to refine those programs until a solution program is found. To determine which program to be refined, WorldCoder uses balances between applying the action of refining on the most likely succesful program and exploring on other programs by using Thompson sampling.

CWM, on the other hand, defines a process they refer to as GIF-MCTS. This process uses a standard Monte Carlo tree search (MCTS), and either prompts an LLM to *generate* a new code world model on a new leaf node, or identifies an issue with the code on an existing leaf node. If the code runs successfully, but does not correctly evaluate the environment, the LLM is prompted to *improve* the code such that the evaluation is more successful. If the code crashes, the code and its error is passed to the LLM with a prompt to *fix* the broken code.

While WorldCoder and CWM are designed for deterministic environments, Poe-World can model the stochasticity by building a collection of codes and then combining them through a learnt set of weights.

LLM-based world model evaluation. Traditionally, world model is evaluated by the accuracy of next state, next reward and next termination prediction (8) (13). In high-stakes domains, besides above predictions, world model also needs to know if the action is valid (performable) or not (11). (12) argue that instead of thinking world model as a generic simulator, its evaluation should focus on predictions relevant to the desired policy. This results in three policy-conditioned tasks: 1) policy verification: check if the world model can induce correct trajectories generated by a policy, 2) action proposal: check if the world model can propose observed action sequence given states generated by a policy, and 3) policy planning: check if we can derive a desired policy by planning with the world model. (14) measure the statistical difference between true value in the environment and the estimated value of a policy in the world model by different metrics: 1) absolute difference, 2) rank correlation, and 3) regret@k.

5. Conclusion

We set out to determine whether an LLM-based code world model generation method could produce accurate models for language grounding-based world model tasks, and if so, attempt to derive a successful and efficient policy using these world models. Current methods of code world model generation using LLMs do not seem to be able to handle descriptive language as part of their observation space, and these natural sentences must be parsed into discrete data points in order to be usable by the generated world models. Additionally, policy generation for language grounding problems creates additional complexity, as maintaining a history of grounded terms (such as grounding an entity's name to its discrete ID in the code world model) does not always seem to be possible for the policy generation methods used by existing LLM-generated code world model systems.

References

[1] Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. Generating Code World Models with Large Language Models Guided by Monte Carlo Tree Search, Oct. 2024. arXiv:2405.15383 [cs]. 1, 2, 4, 5

[2] Noah Goodman, Vikash Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models, July 2014. arXiv:1206.3255 [cs]. 5

[3] Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging Pre-trained Large Language Models to Construct and Utilize World Models for Model-based Task Planning, May 2023. arXiv:2305.14909 [cs]. 5

[4] Austin W. Hanjie, Victor Zhong, and Karthik Narasimhan. Grounding Language to Entities and Dynamics for Generalization in Reinforcement Learning, June 2021. arXiv:2101.07393 [cs]. 1, 2, 3, 4, 7

[5] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder Technical Report, Nov. 2024. arXiv:2409.12186 [cs]. 4

[6] Wasu Top Piriyakulkij, Yichao Liang, Hao Tang, Adrian Weller, Marta Kryven, and Kevin Ellis. PoE-World: Compositional World Modeling with Products of Programmatic Experts, May 2025. arXiv:2505.10819 [cs]. 2, 5

[7] Hao Tang, Darren Key, and Kevin Ellis. WorldCoder, a Model-Based LLM Agent: Building World Models by Writing Code and Interacting with the Environment, May 2024. arXiv:2402.12275 [cs]. 1, 2, 5

[8] Ruoyao Wang, Graham Todd, Ziang Xiao, Xingdi Yuan, Marc-Alexandre Côté, Peter Clark, and Peter Jansen. Can Language Models Serve as Text-Based World Simulators?, June 2024. arXiv:2406.06485 [cs]. 5

[9] Lionel Wong, Gabriel Grand, Alexander K. Lew, Noah D. Goodman, Vikash K. Mansinghka, Jacob Andreas, and Joshua B. Tenenbaum. From Word Models to World Models: Translating from Natural Language to the Probabilistic Language of Thought, June 2023. arXiv:2306.12672 [cs]. 5

[10] Lionel Wong, Jiayuan Mao, Pratyusha Sharma, Zachary S Siegel, Jiahai Feng, Noa Korneev, Joshua B Tenenbaum, and Jacob Andreas. LEARNING ADAPTIVE PLANNING REPRESENTATIONS WITH NATURAL LANGUAGE GUIDANCE. 2024. 5

[11] Kaige Xie, Ian Yang, John Gunerli, and Mark Riedl. Making Large Language Models into World Models with Precondition and Effect Knowledge, Oct. 2024. arXiv:2409.12278 [cs]. 5

[12] Chang Yang, Xinrun Wang, Junzhe Jiang, Qinggang Zhang, and Xiao Huang. Evaluating World Models with LLM for Decision Making, Nov. 2024. arXiv:2411.08794 [cs]. 5

[13] Alex Zhang, Khanh Nguyen, Jens Tuyls, Albert Lin, and Karthik Narasimhan. Language-Guided World Models: A Model-Based Approach to AI Control, July 2024. arXiv:2402.01695 [cs]. 5

[14] Zhilong Zhang, Ruifeng Chen, Junyin Ye, Yihao Sun, Pengyuan Wang, Jingcheng Pang, Kaiyuan Li, Tianshuo Liu, Haoxin Lin, Yang Yu, and Zhi-Hua Zhou. WHALE: Towards Generalizable and Scalable World Models for Embodied Decision-making, Nov. 2024. arXiv:2411.05619. 5

6. Appendix

6.1. MESSENGER details

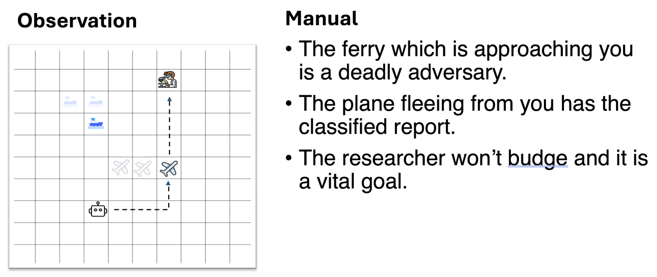
MESSENGER provides three stages with different levels of language generalization assessment:

1. **Stage 1 (S1)**: This stage tests the agent the ability to ground entity name in the manual to entity symbols in the observation, e.g. given the sentence: "The plane has the classified report.", the agent needs to ground this sentence to entity symbol ✈ through the entity name `the plane` and the scalar game rewards, and thus finding that it is a `messenger`. In test games, the objective for the agent is to generalize over 1) new languages describing the same entity name using synonyms, e.g. `researcher-scholar` and 2) new languages describing new combinations of known entities in a game, i.e. the agent has played with entities `ferry`, `plane`, `researcher` in train but not in the same game, and the agent is tasked to play with all of them in a test game.

   As shown in Figure 1, this stage includes three entities, each with one of the three roles: `enemy`, `messenger`, and `goal`, along with their corresponding descriptions. All entities are stationary and placed two steps away from the agent, which starts in the center of the grid. The language descriptions only specify the entities and their roles, with no mention of movement. The agent starts the game either with or without the `message`.

2. **Stage 2 (S2) and S2-dev**: As shown in Figure **??** [6], S2 uses the same set of entities as S1 but introduces movement dynamics: entities can now exhibit

[6]This figure is reproduced from Figure **??** and included here for clarity and the natural flow of the discussion.

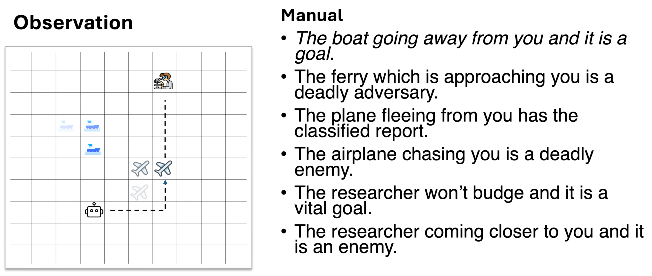


- The ferry which is approaching you is a deadly adversary.
- The plane fleeing from you has the classified report.
- The researcher won't budge and it is a vital goal.

Figure 2. An example of a game play instantiated by MESSENGER S2 (4) within a 10 × 10 grid-world. The observation on the left includes three entities (`ferry`, `plane`, `researcher`) and one agent (depicted by). The game involves three roles: `messenger`, `goal`, and `enemy`. These roles are not explicitly indicated in the observation but are described in the language manual (on the right). The agent's task is to identify their roles, locate the `messenger`, deliver it to the `goal`, and avoid the `enemy`. To achieve this objective, the agent must use the manual to infer entity roles based on their described dynamics and observed behavior. In the observation in the example, shaded icons indicate one possible scenario of entity locations over time. After inferring all entity roles, the agent can execute an appropriate plan to complete the task. The dashed line in the observation shows such a possible plan.

one of three movement types: `moving`, `fleeing`, or `stationary`. The agent always starts without the `message`. During training games, only *one* movement combination is used: one `moving` (chasing), one `fleeing`, and one `stationary` entity, all of which describe how entities are moving compared to the agent. In test games, the agent must handle scenarios where a movement type can appear multiple times, e.g. `moving-moving-fleeing`. To examine the impact of this single-movement constraint, MESSENGER provides a different stage S2-dev, a variation of S2 that also features unseen dynamics but maintains the same movement constraint observed during training for all test games.

In addition to the capabilities demonstrated in S1, the objective of the agent in S2 is to generalize across new language featuring novel environmental dynamics. Specifically, the agent must understand the movement descriptions to make optimal actions, but does not need to ground movement descriptions to the entities based on their observed behaviors. This is because the agent can ground the sentences to the entities based on their names in the manual and their associated symbols in the observation. For example, given the game in Figure 2, the agent can ground the sentence "The plane fleeing from you has the classified report" to entity symbol based on the entity name `the plane-to-symbol` mapping. The agent must understand the entity's behavior to move closer to it, and it can achieve this based on the description "The plane fleeing from you", even without directly observing the behavior.



- *The boat going away from you and it is a goal.*
- The ferry which is approaching you is a deadly adversary.
- The plane fleeing from you has the classified report.
- The airplane chasing you is a deadly enemy.
- The researcher won't budge and it is a vital goal.
- The researcher coming closer to you and it is an enemy.

Figure 3. An example game of MESSENGER S3. To win the game, the agent must infer the roles of entities given the manual. Specifically, the same entity names (e.g. `airplane`, `plane` with different roles (e.g. `enemy`, `messenger`) must be disambiguated by their movement dynamics (e.g. `chasing`, `fleeing`). Note that we have a *italicized* sentence describing an extraneous entity that is not available in the game observation. We also have synonyms for entity names and roles, e.g., `airplane`, `plane`; `adversary`, `enemy`. The shaded entities show possible entity locations over time and the dashed line shows a possible path for the agent to win the game.

3. **Stage 3 (S3)**: In addition to the capabilities demonstrated in Stage 1, the objective of the agent in Stage 3 is to generalize over new language featuring new combinations of known entity movement dynamics. Unlike in Stage 2, the agent in S3 must ground the sentences using both entity name-to-symbol mappings and observed entity behavior-to-movement description mappings.

   As shown in Figure 3, this stage includes five entities, three retain the roles of `enemy`, `messenger`, and `goal`. The manual has six sentences featuring these five entities and one extraneous entity, which is not available in the observation. Specifically, its referred sentence has the same name as the `enemy` entity, but is different in movement and is described as either `goal` or `messenger`.

   Two additional entities are duplicates that share the same entity symbols and names of the `messenger` and `goal` accordingly, but they are assigned the role of `enemy`. To differentiate these entities, their movement dynamics must be used [7]. For example, descriptions like "the fleeing enemy is the dog" versus "the chasing goal is the dog" help the agent identify the correct entities based on their behaviors. In this case, the

[7]The readers are encouraged to read the example in Figure 3 to find the roles of each entity in the observation to understand how the agent reads to manual to complete the task.

dog that is consistently going towards the agent can be inferred as the goal.

6.2. Prompts used for CWM with language grounding (CWM w/ LG)

We follow the same prompt for all three actions in CWM. Please refer to the paper to see the prompts.

This is the prompt we use to describe the environment information of MESSENGER S1.

```
This is the description of MESSENGER stage S1.

## Rules

You control an avatar on a grid (10 x 10) populated by three entities, each
   represented by a unique symbol.
One entity is the enemy, another is the messenger, and the last is the goal.
These entities do not move (only the avatar can move).
Your avatar can move orthogonally on this grid, and can interact with an entity by
    occupying the same space on the grid as an entity.

There are two possible inital states for the avatar:

- with the message: the goal is to deliver the message to the goal entity
- without the message: the goal is to go to the message entity

There are twelve different entities denoted by a fixed set of corresponding entity
    ids that are used consistently across game instances:

Name are the following: = [
"airplane", "mage", "dog", "bird", "fish", "scientist", "thief", "ship", "ball", "
   robot", "queen", "sword", 'wall]

Here is the correct entity id to entity name mapping:
0: background
1: dirt
2: airplane
3: mage
4: dog
5: bird
6: fish
7: scientist
8: thief
9: ship
10: ball
11: robot
12: queen
13: sword
14: wall
avatar_id= 15: no_message
avatar_id=16: with_message

### NOTE about entity id to entity name grounding:

Note that the observation does not have entity names (e.g. "airplane") and the
   agent must observe the entity's symbol and ground the entity's name to its
   corresponding entity id from the manual, based on the reward the agent get. For
    example, if the agent with the message gets a reward of 1 when an entity id X,
    it means that the entity id X is the goal entity. The agent then use the
   manual to ground the entity's name to its corresponding entity id. For example,
    if the manual says "{"name": "dog", "role": "goal"}" and entity id 1 is the
   goal, the agent will ground the entity's name to its corresponding entity id 1:
    "dog" -> 1.

The mapping between entity id (e.g. 1) and entity name (e.g. cat) are consistent
   across all games. However, the mapping between entity id / entity name and
   entity role (e.g. goal, message, enemy) is different across games. You have to
   find the first mapping across all games to infer the second mapping for each
   game. See the example below for more details how to solve this.

## The objective of the game

You win the game by achieving the following mission, depending on the avatar's
   initial state:

- if the avatar is in the initial state with the message: controlling the avatar
   to deliver the message to the goal entity

- if the avatar is in the initial state without the message: controlling the
   avatar to go to the messenger entity
 You have to avoid the enemy entity at all times regardless of the avatar's
     initial state.

Once you achieve the mission, the game is done immediately and you receive the
   reward of 1.

You lose the game if you interact with the enemy entity or you fail to achieve the
    mission before the game ends (done=True).
You can also lose the game if you interact with the wrong entity. E.g. if the
   avatar with message interacts with the messenger entity or the avatar without
   message interacts with the goal entity.
When lose the game, the game ends immediately and you receive the reward of -1.

## Action Space

The action is a `ndarray` with shape `(1,)` which can take values `{0, 1, 2, 3,
   4}` indicating the direction you wish to move your avatar.

- 0: Move up # x = x - 1
- 1: Move down # x = x + 1
- 2: Move left # y = y - 1
- 3: Move right # y = y + 1
- 4: Do not move

## Observation Space

The observation is a Python dictionary with 5 entries corresponding to the
   following `ndarray` elements:

- 0: entity_ids:
- 1: entity_pos
- 2: avatar_ids
- 3: avatar_pos
- 4: manuals
- 5: time

entity_ids is a `ndarray` with shape `(3,)` with values corresponding to the ids
   of the three entities in the environment. Each value in entity_ids corresponds
   to a different entity. The range of entity_ids is [1-14]. Note that one entity
   always has the same entity id across all games. Entity ids does not change over
    time.

entity_pos is a `ndarray` with shape `(3,2)` with values corresponding to the x
   and y coordinates of the three entities on the grid. The x and y coordinates
   range from [0, 9]. Entity pos does not change over time.

avatar_ids is a `ndarray` with shape `(1,)` with a value corresponding to the id
   of your avatar. This value corresponds to the same entities as the table used
   for entity_ids. The avatar can have messenge, then avatar_ids is [16].
   Otherwise, avatar_ids is [15] (without message). Note that avatar_ids does not
   change over time, even when the avatar without message gets the message from
   the messenger entity, avatar_ids is still [15].

avatar_pos is a `ndarray` with shape `(2,)` with values corresponding to the x and
    y coordinates of your avatar. The x and y coordinates range from [0, 9].

time is the number of steps the avatar has taken. Time changes over time. Each
   time step increases by 1. Start from 0 (reset state). Time is from [0, 4]

manuals is the parsed manuals and it is list of dictionaries where each dictionary
    has the following keys:

- name: entity name
- role: entity role (enemy, message (denoted for messenger), or goal)
 e.g. {"name": "dog", "role": "goal"}

The manuals are UNORDERED. Meaning that the first manual does not always
   correspond to the first entity in the entity_ids array. We already shuffled the
    manuals for each game instance.

Entity interacts with the avatar if they share the same position (x, y) on the
   grid, meaning entity position entity_pos == avatar_pos.

Note that the order in all the ndarrays doesn't matter and doesn't reflect the
   role of the entities. For example, entity_ids = [4, 5, 6] might be the enemy,
   message and goal in one game, but it could also be the goal, message and enemy
   entities in other games.
Because the agent only moves, the only information that changes over time is
   avatar_pos and time.

## Rewards

If you win the game by completing the mission, the reward is 1.
If you lose the game by interacting with the enemy entity or interacting with the
   wrong entity, the reward is -1
When the game ends (done=True), if you fail to complete the mission, the reward is
    -1.
```

```
Otherwise, before the game ends, the reward is always 0.

## Starting State

The enemy, message, and goal entities are assigned one of four random starting
    positions: (3,5), (5,3), (5,7), and (7,5). The role of these entities is
    unknown.
The avatar is assigned a starting position of (5,5). The avatar can have message
    or not.

## Episode End

The episode ends (done=True) if any one of the following occurs:

1. Termination: The avatar interacts with ANY entity. Note that in this game, the
    avatar only interacts with just one entity and then the game is terminated
    immediately.
2. Truncation: The time is equal to 4 (avatar has taken 4 steps).

# Class Definition

The class is called "Environment". It has at least the following functions:

- an **init** function to set up the Environment, Keep the mapping entity id to
    entity name in here. And other information that you need to solve the game.

- a "set_state" function to set the current observation of the environment. The
    input parameters for the set_state function is the dictionary of the
    observation.

- a "step" function to predict the next observation in the environment given the
    current observation and an action. The input parameter for the step function is
     an action represented with an integer representing the direction in which the
    player will move.

The outputs of the step function are the TUPLE of :

- the next observation, which has the same format as the observation. Make sure
    the shape of the next observation is the same as the observation.
- the reward of the next observation.
- a boolean variable indicating if the game is done or not
- None variable (for extra information)

### Notes about how to solve this class

Given the rules and description above, you can infer the next observation, next
    reward, and next done from the current observation and the action.

Note that you can't store all manuals in the environment class, because the
    manuals are different for each game instance.

You have to read the manual from the observation and understand the roles of the
    entities, then use this role information and the manual to ground the entity's
```

```
   name to its corresponding entity id. For example, if the manual says {"name": "
   dog", "role": "goal"} and you see agent interacts with entity id 1, get the
   reward of 1, then you can infer that entity id 1 is the goal entity. You can
   also infer entity id 1 is the dog (based on the manual).Therefore, the dog is
   only entity id 1 across all games.

## Example of two episodes / trajectories in MESSENGER S1

#### First episode

#### First observation

entity_ids = [4, 5, 6]
entity_pos = [[3, 5], [5, 3], [5, 7]]
avatar_pos = [5, 5]
avatar_ids = [15]
manuals = [{"name": "dog", "role": "messenger"}, {"name": "fish", "role": "goal"},
    {"name": "bird", "role": "enemy"}]
reward = 0
done = False

In this episode, the avatar is without the message, because the avatar_ids is
   [15]. The agent doesn't know the role of the entities, but it can infer the
   role of the entities based on the reward in the future.

#### Next action

action = 0 # avatar move up

#### Next observation

entity_ids = [4, 5, 6]
entity_pos = [[3, 5], [5, 3], [5, 7]]
avatar_pos = [4, 5]
avatar_ids = [15]
manuals = [{"name": "dog", "role": "message"}, {"name": "fish", "role": "goal"},
   {"name": "bird", "role": "enemy"}]
reward = 0
done = False

#### Next action

action = 0 # avatar move up

#### Next observation

entity_ids = [4, 5, 6]
entity_pos = [[3, 5], [5, 3], [5, 7]]
avatar_pos = [3, 5]
avatar_ids = [15]
manuals = [{"name": "dog", "role": "message"}, {"name": "fish", "role": "goal"},
   {"name": "bird", "role": "enemy"}]
reward = 1
```

```
done = True

#### Explanation:

Given entity-id-to-name mapping above, the agent can infer that entity id 4 is the
     dog. Given the manual, the agent can infer that entity dog with entity id 4 is
     the messenger.

The agent goes up and interacts with entity id 4, because they share the same
    position. The entity id 4 is the dog, and the dog is the messenger. The agent
    gets the reward of 1, because the dog is the messenger and the agent initial
    state is without the message.

The agent therefore wins the game, receives the reward of 1, and the game is done.

#### Second episode

#### First observation

entity_ids = [4, 5, 6]
entity_pos = [[3, 5], [5, 3], [5, 7]]
avatar_pos = [5, 5]
avatar_ids = [15]
manuals = [{"name": "dog", "role": "message"}, {"name": "fish", "role": "goal"},
    {"name": "bird", "role": "enemy"}]
reward = 0
done = False

In this episode, the avatar is without the message, because the avatar_ids is
    [15]. The agent doesn't know the role of the entities, but it can infer the
    role of the entities based on the reward in the future.

#### Next action

action = 3 # avatar move right

#### Next observation

entity_ids = [4, 5, 6]
entity_pos = [[3, 5], [5, 3], [5, 7]]
avatar_pos = [5, 6]
avatar_ids = [15]
manuals = [{"name": "dog", "role": "message"}, {"name": "fish", "role": "goal"},
    {"name": "bird", "role": "enemy"}]
reward = 0
done = False

#### Next action

action = 3 # avatar move right

#### Next observation
```

```
entity_ids = [4, 5, 6]
entity_pos = [[3, 5], [5, 3], [5, 7]]
avatar_pos = [5, 7]
avatar_ids = [15]
manuals = [{"name": "dog", "role": "message"}, {"name": "fish", "role": "goal"},
   {"name": "bird", "role": "enemy"}]
reward = -1
done = True

#### Explanation:

Given entity-id-to-name mapping above, the agent can infer that entity id 6 is the
    bird. Given the manual, the agent can infer that entity bird with entity id 6
   is the enemy.

The agent goes right and interacts with entity id 6, because they share the same
   position. The entity id 6 is the bird, and the bird is the enemy. The agent
   loses the game, receives the reward of -1, and the game is done.
```

6.3. Prompts used for CWM without language grounding (CWM w/t LG)

To maintain a entity id to name mapping and learn this mapping over time, we ask LLM to include function "save_state" where it saves entity to name mapping to disk. We then reload this state from disk and add this state to prompt when we activate the action "Improve".

```
This is the description of MESSENGER stage S1.

## Rules

You control an avatar on a grid (10 x 10) populated by three entities, each
   represented by a unique symbol.
One entity is the enemy, another is the messenger, and the last is the goal.
These entities do not move (only the avatar can move).
Your avatar can move orthogonally on this grid, and can interact with an entity by
    occupying the same space on the grid as an entity.

There are two possible inital states for the avatar:

- with the message: the goal is to deliver the message to the goal entity
- without the message: the goal is to go to the message entity

There are twelve different entities denoted by a fixed set of corresponding entity
    ids that are used consistently across game instances:

Name are the following: = [
"airplane", "mage", "dog", "bird", "fish", "scientist", "thief", "ship", "ball", "
   robot", "queen", "sword", 'wall]

avatar_id= 15: no_message
avatar_id=16: with_message

### NOTE about entity id to entity name grounding:

So basically, you have to infer the role of the entities based on the reward,
    entity ids and entity_pos, and the rule of the game. Then using manual, you can
     infer entity id to entity name mapping. Using this mapping, you can predict
    the next state of the environment and reward in the future. Save this mapping
    in the environment class to use it in the future.

Note that the observation does not have entity names (e.g. "airplane") and the
    agent must observe the entity's symbol and ground the entity's name to its
    corresponding entity id from the manual, based on the reward the agent get. For
     example, if the agent with the message gets a reward of 1 when an entity id X,
     it means that the entity id X is the goal entity. The agent then use the
    manual to ground the entity's name to its corresponding entity id. For example,
     if the manual says "{"name": "dog", "role": "goal"}" and entity id 1 is the
    goal, the agent will ground the entity's name to its corresponding entity id 1:
     "dog" -> 1.

The mapping between entity id (e.g. 1) and entity name (e.g. cat) are consistent
    across all games. However, the mapping between entity id / entity name and
    entity role (e.g. goal, message, enemy) is different across games. You have to
    find the first mapping across all games to infer the second mapping for each
    game. See the example below for more details how to solve this.

## The objective of the game

You win the game by achieving the following mission, depending on the avatar's
    initial state:

- if the avatar is in the initial state with the message: controlling the avatar
    to deliver the message to the goal entity

- if the avatar is in the initial state without the message: controlling the
    avatar to go to the messenger entity
  You have to avoid the enemy entity at all times regardless of the avatar's
      initial state.

Once you achieve the mission, the game is done immediately and you receive the
    reward of 1.

You lose the game if you interact with the enemy entity or you fail to achieve the
     mission before the game ends (done=True).
You can also lose the game if you interact with the wrong entity. E.g. if the
    avatar with message interacts with the messenger entity or the avatar without
    message interacts with the goal entity.
When lose the game, the game ends immediately and you receive the reward of -1.

## Action Space

The action is a `ndarray` with shape `(1,)` which can take values `{0, 1, 2, 3,
    4}` indicating the direction you wish to move your avatar.

- 0: Move up # x = x - 1
- 1: Move down # x = x + 1
- 2: Move left # y = y - 1
- 3: Move right # y = y + 1
- 4: Do not move

## Observation Space

The observation is a Python dictionary with 5 entries corresponding to the
    following `ndarray` elements:

- 0: entity_ids:
- 1: entity_pos
- 2: avatar_ids
- 3: avatar_pos
- 4: manuals
- 5: time

entity_ids is a `ndarray` with shape `(3,)` with values corresponding to the ids
    of the three entities in the environment. Each value in entity_ids corresponds
    to a different entity. The range of entity_ids is [1-14]. Note that one entity
    always has the same entity id across all games. Entity ids does not change over
     time.

entity_pos is a `ndarray` with shape `(3,2)` with values corresponding to the x
    and y coordinates of the three entities on the grid. The x and y coordinates
    range from [0, 9]. Entity pos does not change over time.

avatar_ids is a `ndarray` with shape `(1,)` with a value corresponding to the id
    of your avatar. This value corresponds to the same entities as the table used
    for entity_ids. The avatar can have messenge, then avatar_ids is [16].
    Otherwise, avatar_ids is [15] (without message). Note that avatar_ids does not
    change over time, even when the avatar without message gets the message from
    the messenger entity, avatar_ids is still [15].

avatar_pos is a `ndarray` with shape `(2,)` with values corresponding to the x and
     y coordinates of your avatar. The x and y coordinates range from [0, 9].

time is the number of steps the avatar has taken. Time changes over time. Each
    time step increases by 1. Start from 0 (reset state). Time is from [0, 4]

manuals is the parsed manuals and it is list of dictionaries where each dictionary
     has the following keys:

- name: entity name
- role: entity role (enemy, message (denoted for messenger), or goal)
  e.g. {"name": "dog", "role": "goal"}

The manuals are UNORDERED. Meaning that the first manual does not always
    correspond to the first entity in the entity_ids array. We already shuffled the
     manuals for each game instance.

Entity interacts with the avatar if they share the same position (x, y) on the
    grid, meaning entity position entity_pos == avatar_pos.

Note that the order in all the ndarrays doesn't matter and doesn't reflect the
    role of the entities. For example, entity_ids = [4, 5, 6] might be the enemy,
    message and goal in one game, but it could also be the goal, message and enemy
    entities in other games.
Because the agent only moves, the only information that changes over time is
    avatar_pos and time.

## Rewards

If you win the game by completing the mission, the reward is 1.
If you lose the game by interacting with the enemy entity or interacting with the
    wrong entity, the reward is -1
When the game ends (done=True), if you fail to complete the mission, the reward is
     -1.
Otherwise, before the game ends, the reward is always 0.

## Starting State

The enemy, message, and goal entities are assigned one of four random starting
    positions: (3,5), (5,3), (5,7), and (7,5). The role of these entities is
    unknown.
The avatar is assigned a starting position of (5,5). The avatar can have message
    or not.

## Episode End

The episode ends (done=True) if any one of the following occurs:

1. Termination: The avatar interacts with ANY entity. Note that in this game, the
    avatar only interacts with just one entity and then the game is terminated
    immediately.
2. Truncation: The time is equal to 4 (avatar has taken 4 steps).

# Class Definition

The class is called "Environment". It has at least the following functions:

- an **init** function to set up the Environment, Keep the mapping entity id to
    entity name in here, you will provide it in the prompt later if it exists. And
    other information that you need to solve the game.

- a "set_state" function to set the current observation of the environment. The
    input parameters for the set_state function is the dictionary of the
    observation.

- a "step" function to predict the next observation in the environment given the
    current observation and an action. The input parameter for the step function is
     an action represented with an integer representing the direction in which the
    player will move.

- a "save_state" function where you save entity-id to entity name mapping to disk
    in "states" folder.

The outputs of the step function are the TUPLE of :
```

```
- the next observation, which has the same format as the observation. Make sure
    the shape of the next observation is the same as the observation.
- the reward of the next observation.
- a boolean variable indicating if the game is done or not
- None variable (for extra information)

### Notes about how to solve this class

Given the rules and description above, you can infer the next observation, next
    reward, and next done from the current observation and the action.

Note that you can't store all manuals in the environment class, because the
    manuals are different for each game instance.

You have to read the manual from the observation and understand the roles of the
    entities, then use this role information and the manual to ground the entity's
    name to its corresponding entity id. For example, if the manual says {"name": "
    dog", "role": "goal"} and you see agent interacts with entity id 1, get the
    reward of 1, then you can infer that entity id 1 is the goal entity. You can
    also infer entity id 1 is the dog (based on the manual). Therefore, the dog is
    only entity id 1 across all games.

## Example of two episodes / trajectories in MESSENGER S1

#### First episode

#### First observation

entity_ids = [4, 5, 6]
entity_pos = [[3, 5], [5, 3], [5, 7]]
avatar_pos = [5, 5]
avatar_ids = [15]
manuals = [{"name": "dog", "role": "messenger"}, {"name": "fish", "role": "goal"},
     {"name": "bird", "role": "enemy"}]
reward = 0
done = False

In this episode, the avatar is without the message, because the avatar_ids is
    [15]. The agent doesn't know the role of the entities, but it can infer the
    role of the entities based on the reward in the future.

#### Next action

action = 0 # avatar move up

#### Next observation

entity_ids = [4, 5, 6]
entity_pos = [[3, 5], [5, 3], [5, 7]]
avatar_pos = [4, 5]
avatar_ids = [15]
manuals = [{"name": "dog", "role": "message"}, {"name": "fish", "role": "goal"},
    {"name": "bird", "role": "enemy"}]
```

```
reward = 0
done = False

#### Next action

action = 0 # avatar move up

#### Next observation

entity_ids = [4, 5, 6]
entity_pos = [[3, 5], [5, 3], [5, 7]]
avatar_pos = [3, 5]
avatar_ids = [15]
manuals = [{"name": "dog", "role": "message"}, {"name": "fish", "role": "goal"},
    {"name": "bird", "role": "enemy"}]
reward = 1
done = True

#### Explanation:

The agent then looks at entity ids and realizes that entity id 4 is the entity it
    interacts with.
THe agent sees the reward of 1 and beacause it does not have the message, it can
    infer that the entity it interacts with is the messenger. THe agent then reads
    the manual and see that the dog is the messenger.

Therefore entity id 4 is the messenger, and is also the dog. The agent then
    remembers this mapping: 4<->dog. In the future, using this mapping, the agent
    can infer the role of the dog based on the manual and predict rewards
    appropriately.

#### Second episode

#### First observation

entity_ids = [4, 5, 6]
entity_pos = [[3, 5], [5, 3], [5, 7]]
avatar_pos = [5, 5]
avatar_ids = [15]
manuals = [{"name": "dog", "role": "message"}, {"name": "fish", "role": "goal"},
    {"name": "bird", "role": "enemy"}]
reward = 0
done = False

In this episode, the avatar is without the message, because the avatar_ids is
    [15]. The agent doesn't know the role of the entities, but it can infer the
    role of the entities based on the reward in the future.

#### Next action

action = 3 # avatar move right

#### Next observation
```

```
entity_ids = [4, 5, 6]
entity_pos = [[3, 5], [5, 3], [5, 7]]
avatar_pos = [5, 6]
avatar_ids = [15]
manuals = [{"name": "dog", "role": "message"}, {"name": "fish", "role": "goal"},
    {"name": "bird", "role": "enemy"}]
reward = 0
done = False

#### Next action

action = 3 # avatar move right

#### Next observation

entity_ids = [4, 5, 6]
entity_pos = [[3, 5], [5, 3], [5, 7]]
avatar_pos = [5, 7]
avatar_ids = [15]
manuals = [{"name": "dog", "role": "message"}, {"name": "fish", "role": "goal"},
    {"name": "bird", "role": "enemy"}]
reward = -1
done = True

#### Explanation:

The agent understand that the entity it interacts with is entity 6, because they
    share the same position.

The agent sees the reward -1 and realizes that it interacted with the enemy.

Therefore, entity id 6 is the enemy. The agent then reads the manual and sees that
     the bird is the enemy.

Therefore, entity id 6 is the enemy, and is also the bird. The agent then
    remembers this mapping: 6<->bird. In the future, using this mapping, the agent
    can infer the role of the bird based on the manual and predict rewards
    appropriately.

### NOTES about these examples

Don't remember these examples, the dog, bird, fish, etc. can be any entity id
    [0-14] in real games. The agent then can keep this entity id to entity name
    mapping for future games, to infer the role of the entities, which can be
    changed across games, but not entity id to entity name mapping.

IMPORTANT: (parsed version or real sentence) manual in manuals does not follow the
     order in entity_ids or entity_pos array. Meaning that e.g. the first manual
    does not always correspond to the first entity in the entity_ids array. We
    already shuffled the manuals for each game instance.

Through many trajectories, given rewards, the agent can learn to ground the entity
```

```
    's name to its corresponding entity id. -->
```

6.4. World model code for CWM without language grounding (CWM w/t LG

```
import numpy as np
import os
import json


class Environment:
    def __init__(self):
        # Fixed entity id to entity name mapping (to be inferred during play)
        self.entity_id_to_name = {}
        # Will be filled after first observation
        # entity_id -> role ('enemy', 'message', 'goal')
        self.role_mapping = {}
        self.entity_ids = None
        self.entity_pos = None
        self.avatar_id = None
        self.avatar_pos = None
        self.manuals = None
        self.time = 0
        self.done = False
        self.reward = 0
        self.state = None  # current observation
        # For simplicity, define walls if needed, but as per description, walls
            are entities with id 'wall'
        self.walls = []  # optional, not specified in the problem
        # Initialize entity id to name mapping as unknown
        self.known_entity_id_to_name = {}  # to be inferred
        # For the first run, roles are unknown
        self.roles_inferred = False

    def set_state(self, observation):
        # Set current environment state from observation
        self.state = observation
        self.entity_ids = observation['entity_ids']
        self.entity_pos = observation['entity_pos']
        self.avatar_id = observation['avatar_ids'][0]
        self.avatar_pos = observation['avatar_pos']
        self.manuals = observation['manuals']
        self.time = observation['time']
        self.done = False
        self.reward = 0
        # Reset role inference if needed
        if not self.roles_inferred:
            self._infer_entity_names()

    def _infer_entity_names(self):
        # Use manuals and reward to infer entity id to name mapping
        # This is an initial inference based on the first observation
        # Since at start, roles are unknown, we will update roles after
            interactions
        self.unknown_roles = {eid: None for eid in self.entity_ids}
        self.roles_inferred = False

    def _update_roles(self, entity_id, role):
        # Store role for entity_id
        self.role_mapping[entity_id] = role
        # Once all roles are known, we can consider roles inferred
        if len(self.role_mapping) == len(self.entity_ids):
            self.roles_inferred = True

    def _get_entity_role(self, entity_id):
        # Return role if known
        return self.role_mapping.get(entity_id, None)

    def _ground_entity_id_to_name(self, entity_id):
        # Ground entity id to name based on manuals and known roles
        role = self._get_entity_role(entity_id)
        if role:
            for manual in self.manuals:
                if manual['role'] == role:
                    return manual['name']
        else:
            # If role not known, return None
            return None

    def step(self, action):
        if self.done:
            # If already done, return current state
            return self.state, self.reward, self.done, None

        # Map action to movement
        move_map = {
            0: (-1, 0),  # up
            1: (1, 0),   # down
            2: (0, -1),  # left
            3: (0, 1),   # right
            4: (0, 0)    # no move
        }
        delta = move_map.get(action, (0, 0))
        new_x = self.avatar_pos[0] + delta[0]
        new_y = self.avatar_pos[1] + delta[1]

        # Check boundaries
        if 0 <= new_x <= 9 and 0 <= new_y <= 9:
            self.avatar_pos = np.array([new_x, new_y])
        # else stay in same position

        self.time += 1

        # Initialize reward and done for this step
        reward = 0
        done = False
    # Check interaction
    interacted_entity_id = None
    for eid, epos in zip(self.entity_ids, self.entity_pos):
        if np.array_equal(epos, self.avatar_pos):
            interacted_entity_id = eid
            break

    if interacted_entity_id is not None:
        role = self._get_entity_role(interacted_entity_id)
        entity_name = self._ground_entity_id_to_name(interacted_entity_id)

        # Determine the role of the entity
        # The initial role inference is based on reward signals
        # We need to assign roles if not known
        if role is None:
            # Infer role based on reward or other heuristics
            # For simplicity, we can assign role based on previous
                interactions
            # but since we don't have that info here, we rely on the reward
                logic below
            pass

        # Check for interaction correctness
        # The environment's rules:
        # - if avatar with message: goal is to reach goal entity
        # - if avatar without message: goal is to reach message entity
        # - interacting with enemy ends game with -1
        # - interacting with wrong entity ends game with -1
        # - interacting with correct entity ends game with +1

        # For now, we try to infer role based on current info:
        # If role is known, use it; otherwise, infer from reward after
            interaction
        # Since at this point, we don't know roles, but we can infer from
            previous info
        # For this implementation, we will use the following logic:

        # Let's define a simple heuristic:
        # If the avatar has message (avatar_id==16):
        #    - interacting with goal: reward=1, done=True
        #    - interacting with enemy: reward=-1, done=True
        #    - interacting with message: reward=-1, done=True (if wrong)
        # If avatar without message (avatar_id==15):
        #    - interacting with message: reward=1, done=True
        #    - interacting with enemy: reward=-1, done=True
        #    - interacting with goal: reward=-1, done=True (if wrong)

        # Since we don't have actual reward signals yet, we implement logic
            based on position:
        # But in this simulation, the reward is assigned after interaction
            based on correctness.
        # So, we simulate the reward logic here:
```

```
# For the purpose of this code, assume that:
# - if the entity is the goal and avatar's message state matches the
    goal's role
# - if the entity is enemy, interaction is negative
# - if the entity is message and avatar has no message, interaction is
     positive

# But since the environment does not store this info explicitly, we
    need to infer from the role and message state:

# Let's implement a simplified logic:
# First, determine if the entity is the goal, message, or enemy based
    on manual info
# We will assign roles if not known, based on the reward signals in
    actual environment
# Here, for simulation, we assume the environment's reward is set
    correctly elsewhere

# For the simulation, we will do:
# - If the entity is the goal and avatar has no message, reward=1
# - If the entity is the message and avatar has message, reward=1
# - If the entity is enemy, reward=-1
# - Else, reward=-1

# For now, since the actual reward is known only after the interaction
    , we will set it accordingly:

# Check if entity is goal, message, or enemy based on manual roles
entity_role = None
for manual in self.manuals:
    if manual['name'] == entity_name:
        entity_role = manual['role']
        break

# Determine reward and done based on roles
if entity_role == 'goal':
    if self._is_with_message():
        # Avatar has message, goal is to deliver message
        reward = 1
    else:
        # Avatar without message interacting with goal is incorrect
        reward = -1
    done = True
elif entity_role == 'message':
    # If avatar doesn't have message, picking message is correct
    if not self._is_with_message():
        self.avatar_id = 16
        reward = 1
    else:
        # Already has message, interacting with message again
        reward = -1
    done = True
elif entity_role == 'enemy':
```

```
                reward = -1
                done = True
            else:
                # Unknown role, treat as wrong interaction
                reward = -1
                done = True

        # Check for time limit
        if self.time >= 4:
            reward = -1
            done = True

        # Save reward and done status
        self.reward = reward
        self.done = done

        # Update state
        self.state = self._get_observation()

        return self.state, reward, done, None

    def _is_with_message(self):
        return self.avatar_id == 16

    def _get_observation(self):
        # Return current observation dict
        return {
            'entity_ids': self.entity_ids,
            'entity_pos': self.entity_pos,
            'avatar_ids': np.array([self.avatar_id]),
            'avatar_pos': self.avatar_pos,
            'manuals': self.manuals,
            'time': self.time
        }
```

6.5. World model code with language grounding: CWM w/ LG

```
import numpy as np


class Environment:
    def __init__(self):
        self.entity_id_to_name = {
            0: "background",
            1: "dirt",
            2: "airplane",
            3: "mage",
            4: "dog",
            5: "bird",
            6: "fish",
            7: "scientist",
            8: "thief",
            9: "ship",
            10: "ball",
            11: "robot",
            12: "queen",
            13: "sword",
            14: "wall",
        }

        self.avatar_id_to_state = {15: "no_message", 16: "with_message"}

        self.name_to_entity_id = {v: k for k,
                                  v in self.entity_id_to_name.items()}
        self.state = {}
        self.done = False
        self.time_limit = 4

    def set_state(self, observation):
        self.state = observation

    def _move_avatar(self, action):
        x, y = self.state["avatar_pos"]
        if action == 0:  # Move up
            new_x, new_y = max(x - 1, 0), y
        elif action == 1:  # Move down
            new_x, new_y = min(x + 1, 9), y
        elif action == 2:  # Move left
            new_x, new_y = x, max(y - 1, 0)
        elif action == 3:  # Move right
            new_x, new_y = x, min(y + 1, 9)
        else:  # Do not move
            new_x, new_y = x, y

        self.state["avatar_pos"] = np.array([new_x, new_y])

    def _determine_reward_and_done(self):
        reward = 0
        self.done = False

        avatar_pos = self.state["avatar_pos"]
        entity_ids = self.state["entity_ids"]
        entity_pos = self.state["entity_pos"]
        avatar_id = self.state["avatar_ids"][0]
        manuals = self.state["manuals"]

        # Create a mapping from entity position to entity id
        entity_position_to_id = {
            tuple(pos): eid for eid, pos in zip(entity_ids, entity_pos)
        }

        # Check if avatar interacts with any entity
        if tuple(avatar_pos) in entity_position_to_id:
            interacted_entity_id = entity_position_to_id[tuple(avatar_pos)]
            interacted_entity_role = next(
                (
                    manual["role"]
                    for manual in manuals
                    if self.name_to_entity_id[manual["name"]] ==
                        interacted_entity_id
                ),
                None,
            )

            if interacted_entity_role == "enemy":
                reward = -1
                self.done = True
            elif avatar_id == 15 and interacted_entity_role == "message":
                reward = 1
                self.done = True
            elif avatar_id == 16 and interacted_entity_role == "goal":
                reward = 1
                self.done = True
            else:
                reward = -1
                self.done = True
        else:
            if self.state["time"] >= self.time_limit:
                reward = -1
                self.done = True

        return reward, self.done

    def step(self, action):
        self._move_avatar(action)
        self.state["time"] += 1
        reward, done = self._determine_reward_and_done()
        return self.state, reward, done, None
```