ONE LIFE TO LEARN: INFERRING SYMBOLIC WORLD MODELS FOR STOCHASTIC ENVIRONMENTS FROM UNGUIDED EXPLORATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Symbolic world modeling is the task of inferring and representing the transitional dynamics of an environment as an executable program. Previous research on symbolic world modeling has focused on simple, deterministic environments with abundant data and human-provided guidance. We address the more realistic and challenging problem of learning a symbolic world model in a complex, stochastic environment with severe constraints: a limited interaction budget where the agent has only "one life" to explore a hostile environment and no external guidance in the form of human-provided, environment-specific rewards or goals. We introduce ONELIFE, a framework that models world dynamics through conditionally-activated programmatic laws within a probabilistic programming framework. Each law operates through a precondition-effect structure, allowing it to remain silent on irrelevant aspects of the world state and predict only the attributes it directly governs. This creates a dynamic computation graph that routes both inference and optimization only through relevant laws for each transition, avoiding the scaling challenges that arise when all laws must contribute to predictions about a complex, hierarchical state space, and enabling accurate learning of stochastic dynamics even when most rules are inactive at any given moment. To evaluate our approach under these demanding constraints, we introduce a new evaluation protocol that measures (a) state ranking, the ability to distinguish plausible future states from implausible ones, and (b) state fidelity, the ability to generate future states that closely resemble reality. We develop and evaluate our framework on Crafter-OO, our reimplementation of the popular Crafter environment that exposes a structured, object-oriented symbolic state and and a pure transition function that operates on that state alone. ONELIFE can successfully learn key environment dynamics from minimal, unguided interaction, outperforming a strong baseline on 16 out of 23 scenarios tested. Our work establishes a foundation for autonomously constructing programmatic world models of unknown, complex environments.[1]

1 INTRODUCTION

World modeling is a critical task in artificial intelligence, providing an agent with a functional understanding of its environment's underlying dynamics. By learning a world model, an agent can predict the outcomes of its actions without having to actually interact with the real world. One line of research in world modeling aims to learn symbolic world models via program synthesis (i.e., representing worlds models with code) with a view towards building representations that are interpretable, editable, and verifiable by humans.

While such approaches have been successful in environments with a limited number of discoverable mechanics and low stochasticity (Piriyakulkij et al., 2025; Tang et al., 2024; Dainese et al., 2024) these assumptions are often violated in more complex environments. Examples of such environments are popular open-world sandbox games (e.g. MineCraft, RuneScape) containing numerous, diverse mechanics spanning crafting, combat, and physics. These more realistic environments have

[1] We will release Crafter-OO and ONELIFE code upon acceptance.

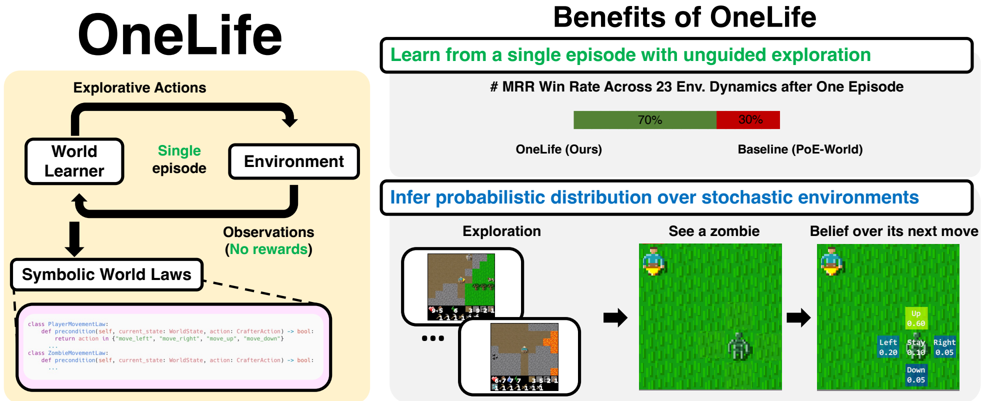

Figure 1: ONELIFE synthesizes world laws from a single unguided (no environment-specific rewards / goals) episode in a hostile, stochastic environment. ONELIFE models the world as mixture of laws in code with a precondition-effect structure, each governing an aspect of the world, and infers parameters for the mixture that best explain the observed dynamics of the world. The resulting world model (WM) provides a probability distribution over attributes of an object-oriented world state, such as the `position` of a particular zombie. ONELIFE outperforms a strong baseline in modeling $16/23$ core game mechanics tested, measured by MRR (Mean Reciprocal Rank) of the true next state (Sec. 4) under the WM's likelihood. See Box A.3 for a synthesized zombie law.

irreducible stochasticity (e.g., outcomes of actions are subject to random chance), a lack of extrinsic rewards (e.g., players set their own goals and there is no well-defined criteria for "winning"), and a high cost of exploration (e.g., entering dangerous areas without preparation can result in death), making it crucial to learn from minimal interaction. This leads to our central research question:

How can an agent reverse engineer the laws of a complex, dangerous stochastic world, given a limited interaction budget and without environment-specific human-specified goals or rewards?

We introduce a framework for symbolic world modeling, ONELIFE, a name that reflects our focus on learning a symbolic world model from a single episode with unguided exploration. As illustrated in Fig. 1 (top-right), ONELIFE learns from just a *single, unguided run* in the environment, a contrast to previous work (Piriyakulkij et al., 2025; Tang et al., 2024; Dainese et al., 2024) that assumes access to a large number of interactions as well as environment specific guidance provided by humans (e.g., goals / rewards designed for the environment). ONELIFE recovers a **program** that describes the environment's underlying transition dynamics $p(s_{t+1}|s_t, a_t)$ which models the probability distribution p over next states s_{t+1} given a current state s_t and action a_t. The agent performs this inference using only observations, **without access to rewards or other domain-specific guidance.** ONELIFE has two key components: **a law synthesizer** (Sec. 3.3) that proposes new laws and **an inference algorithm** (Sec. 3.4) that re-weights laws based on their predictive ability over observations. Crucially, the inference algorithm is gradient-based and only updates the laws that alter the observed variables between current state s_t and predicted next state s_{t+1}, allowing for efficient and targeted learning. These components work together in a probabilistic programming mixture-of-laws approach (Sec. 3.2) that proposes and re-weights rules based on whether the preconditions for the laws to be applicable are met and the effect of the predictions w.r.t. the observed environment transitions. This approach enables our model to infer distributions over complex, stochastic events, as shown in the Fig. 1 (bottom-right), where a learned world model outputs a distribution over a zombie's next move. Crucially, ONELIFE not only produces a distribution over states but *learns from* stochastic observations; the true movement of the zombie in Fig. 1 also follows a distribution, which ONELIFE seeks to approximate.

To evaluate our approach, we first created a suitable testbed — Crafter-OO — by re-engineering the complex Crafter (Hafner, 2022) environment to be a pure function $T(s, a) \rightarrow s'$ of a structured, text-based hierarchical object-oriented world state. In other words, all the information needed to compute the next state is represented in a single structured, object-oriented representation, and there is a ground-truth program for the transition function that computes the next environment state purely from the state representation, without any "hidden variables". This text-based, object-oriented representation is natively readable by LLMs and thus allows them to try reconstructing the transition function by writing code that programatically modifies the structured state. It allows for a structured, object-oriented representation that is directly comprehensible to language models and enables symbolic reasoning over a world with rich entity interactions. We introduce a new evaluation protocol that uses two axes (Sec. 4): **state ranking**, the ability to distinguish valid outcomes from invalid ones according to the world's laws, and **state fidelity**, the ability to produce plausible future states for planning. Our experiments show that ONELIFE better captures the environment's dynamics compared to several baselines, including PoE-World (Piriyakulkij et al., 2025), showing improved ability to simulate future states given a state and candidate action, and to distinguish between likely and unlikely outcomes of an action.

In summary, our contributions include:

- ONELIFE, a probabilistic symbolic world model that can learn from stochastic and hostile environments with minimal interactions and without access to human-defined rewards. ONELIFE outperforms prior work, learning a world model that better predicts true environment dynamics.
- Crafter-OO, a reimplementation of Crafter (Hafner, 2022) that exposes a structured, object-oriented symbolic state and and a pure transition function that operates on that state alone. This enables us to test ONELIFE in a complex, stochastic environment and lays the groundwork for future work in symbolic world modeling and programmatic reinforcement learning.
- An evaluation suite for world modeling within Crafter / Crafter-OO with 30+ executable scenarios that test knowledge of all core mechanics in Crafter and a pool of mutators that can programatically generate illegal distractor states to probe world model understanding alongside, new state fidelity and state ranking metrics for evaluating world models in complex, stochastic, environments.

2 RELATED WORK

Symbolic World Models. Symbolic world models represent an environment's transition dynamics as executable code, producing interpretable, editable, and generalizable models from limited data. Prior work has used LLMs to synthesize a single, monolithic program that functions as a world model (Tang et al., 2024; Dainese et al., 2024). Piriyakulkij et al. (2025) introduced a compositional approach by representing the world model as a product of programmatic experts, enabling modeling of more complex dynamics. Other methods have synthesized programs for planning (Ahmed et al., 2025) or combined functional and automata synthesis to capture latent state dynamics (Das et al., 2023). LLMs have also been used to construct formal planning representations like PDDL from environment interactions or text for symbolic planners (Guan et al., 2023; Deng et al., 2024). Our work differs from these methods in three aspects. First, we operate in a complex, open-world environment based on Crafter (Hafner, 2022) with stochasticity and many interacting mechanics, whereas prior work has operated in simpler, often deterministic domains (e.g., grid-worlds or Atari games). Second, we do not assume abundant interaction data: our agent learns from a limited budget obtained in a single episode – or life. Third, ONELIFE learns without external rewards or human-specified goals, framing the task as unguided reverse engineering of the environment's laws.

Programmatic Representations for Decision-Making. Program synthesis has been used to represent other components of intelligent agents. Programmatic policies have been shown to offer greater interpretability and generalization compared to neural networks (Trivedi et al., 2021; Liang et al., 2022). LLMs have been used to generate programmatic reward functions from natural language instructions, enabling agents to pursue complex, user-specified objectives (Ma et al., 2024; Yu et al., 2023; Klissarov et al., 2025). Programs have been used to build libraries of composable, temporally extended skills, allowing agents to solve long-horizon tasks by combining previously learned behaviors (Wang et al., 2025; Stengel-Eskin et al., 2024). These methods focus on representing components of the agent's internal decision-making process: *how it should act* (policies), *what it should value* (rewards), or *what it is capable of doing* (skills). In contrast, our work learns a model of *how the external world behaves*; this task-agnostic model of environment dynamics is complementary to policies, rewards, and skills, and supports planning and decision-making for any downstream goals.

World Modeling for Open-Ended Exploration and Discovery. Agents that explore and learn in complex, open-world environments without extrinsic rewards typically learn non-symbolic, latent world models and use them to drive exploration through intrinsic motivation (Hafner et al., 2023;

Micheli et al., 2023; Dedieu et al., 2025; Schwarzer et al., 2021). These agents plan using their world models to find novelty or surprise in their environments, discovering useful skills without task-specific supervision (Sekar et al., 2020). This connects to automated scientific discovery, which requires autonomously forming hypotheses and performing experiments to understand unknown systems (Jansen et al., 2024; Chen et al., 2025; Geng et al., 2025). New evaluation frameworks have been proposed to assess an agent's ability to rapidly induce world models in novel contexts (Ying et al., 2025; Vafa et al., 2024). Unlike methods that learn implicit, latent world models, our work learns an explicit, symbolic representation of the world's laws. We frame learning as reverse engineering a complex system's rules from unguided, limited interaction.

3 Overview of OneLife

Our framework, **OneLife** is designed to learn symbolic world models from a single, unguided episode of exploration. It is built on two key abstractions, a programmatic representation of world dynamics as a mixture of modular *laws* with learnable weights and an *observable extractor* that decouples the environment's state from the learning process. The framework consists: a **a world model as a program** (Sec. 3.2), a **law synthesizer** that proposes new laws using offline data from an **unguided exploration policy** (Sec. 3.3), an **inference algorithm** that re-weights laws based on observations (Sec. 3.4), and a **forward simulation process** that uses the learned model for predicting future states (Sec. 3.5).

We model the environment as having a pure, but potentially stochastic, transition function $T : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$, where $\Delta(\mathcal{S})$ is the space of probability distributions over the state space $\mathcal{S}$. This functional view aligns with modern reinforcement learning environment frameworks (Freeman et al., 2021; Matthews et al., 2024) and physical models, where the future state of a system is a pure function of an explicit state and any interventions.

3.1 Crafter-OO: A Testbed for Symbolic World Modeling

A common design assumption in previous work on symbolic world modeling (Tang et al., 2024; Piriyakulkij et al., 2025; Dainese et al., 2024) is that we have access to an object-oriented world state to use as input to the symbolic world model under construction. In practice, this state is only easily accessible for simple environments such as Minigrid (Chevalier-Boisvert et al., 2023) or BabyAI (Chevalier-Boisvert et al., 2018). Programmatic access to the state of more complex environments such as Atari games as used by Piriyakulkij et al. (2025) is only possible due to standalone development efforts such as OCAtari (Delfosse et al., 2024) which makes the internal object-oriented state of these environments accessible to researchers. The lack of an environment with an exposed, object-oriented state that is more complex than gridworlds or with mechanics more diverse than Atari games has thus far prevented evaluation and development of symbolic world modeling approaches for more complex environments. To close this gap, we implement Crafter-OO (Appendix B), which emulates the Crafter (Hafner, 2022) environment by operating purely on an explicit, object-oriented game (Listing 1). Additionally, we contribute utilities for programmatically modifying the game state to create evaluation scenarios (Appendix D, Sec. 5.1).

Our target environment Crafter-OO features significant stochasticity, diverse forms of mechanics, and active non-player characters. This includes elements such as hostile and friendly agents with diverse, inherently random behaviors. Our framework is designed to infer the rules governing these interactions from observation alone, without access to rewards or human-specified goals. For instance, in Fig. 2, the scenario contains a "zombie" character chasing the player via stochastic movements. While one cannot perfectly predict the future position of a zombie due to inherent randomness built into the environment, our world model is able to capture this "chasing the player" behavior without any explicit supervision by predicting a discrete distribution for the `zombie.position` attributes.

3.2 OneLife: World Model as a Mixture of Laws

We consider environments with complex, structured state spaces $\mathcal{S}$ where the full state $s \in \mathcal{S}$ may be hierarchical and contain a mixture of entity types and attributes. An agent interacts with the environment by taking an action $a \in \mathcal{A}$ and observing a transition from state s_t to s_{t+1}, as illustrated in Fig. 2. We model an environment's transition function as a composition of programmatic laws.

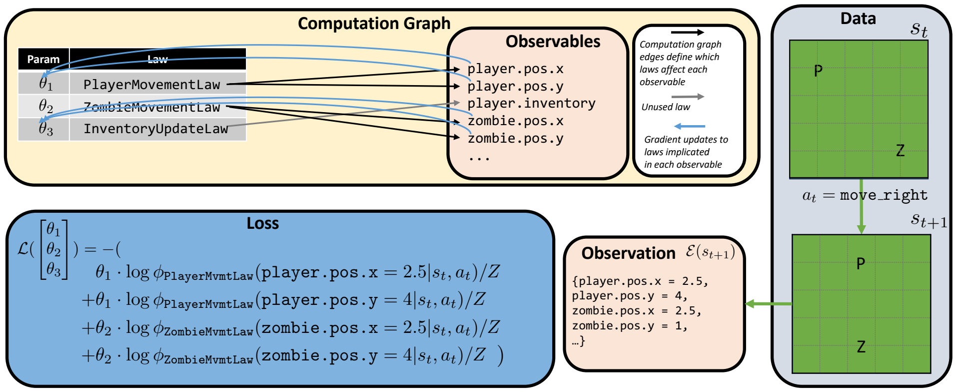

Figure 2: Illustration of the inference process. The active laws for each observable (defined by $\mathcal{I}_k(s_t, a)$) determine the structure of the computation graph, i.e., which laws and their corresponding parameters θ_i are related to which observables. This structure in turn informs the parameter updates. Shown here is a dataset with a single transition instance, in which the player (P) moves right; at the same time, a zombie (Z) independently moves left. this implicates two laws, `PlayerMovementLaw` and `ZombieMovementLaw`, while not implicating the `InventoryUpdateLaw`. As a result, the loss computation is only a function of θ_1 and θ_2. Note we use Z here to denote the normalizing factor. Examples of synthesized laws can be seen in Appendix A.

A law, L_i, is a program defined by a pair (c_i, e_i), where $c_i(s, a) \to \{\text{true}, \text{false}\}$ is a *precondition* and $e_i(s, a) \to s'$ is an *effect*. The precondition determines whether the law is applicable to a state-action pair (s, a). The effect function makes a prediction by modifying attributes on a copy of the state. For example, the `PlayerMovementLaw` in Fig. 2 applies to state-action pairs with a player and a move action, and has an effect on the player position's (x) observable. This precondition-effect structure is inspired by classical planning and provides a natural way to specify the scope of each law, ensuring modularity (McDermott et al., 1998). During any given transition, multiple or no laws may be applicable.

To create a tractable interface to compare states predicted by a world model and the true state of the environment, we introduce an **observable extractor**, $\mathcal{E} : \mathcal{S} \to \mathcal{O}$. This function maps a complex state s into a vector of primitive-valued **observables** $o \in \mathcal{O}$. In the scenario sketched in Fig. 2, the next state s_{t+1} can be complex, with additional entities and objects (e.g., trees, inventory items, etc.). Nevertheless, one can tractably compare states via observations, i.e., *changes* between s_t and s_{t+1} such as `player.position`, `player.inventory`, `zombie.position`, etc. Note that any given law L_i only makes predictions about a subset of all possible observables. For instance, in Fig. 2, the `PlayerMovementLaw` *only* makes predictions about `player.position` observables and *does not predict* the `zombie.position` observables.

Our world model can be viewed as a probabilistic program (van de Meent et al., 2021) that generates the next state's observables o' conditioned on the current state s and action a. The set of laws $\{L_i\}$ defines the components of this program. The effect e_i of each law specifies a set of conditional probability distributions $\phi_{i,o}(o = v|s, a)$ for an observable o, where v denotes a specific outcome in the discrete support of the observable $\text{supp}(o)$. For a given state-action pair (s, a), the set of *active laws* is $\mathcal{I}(s, a) = \{i \mid c_i(s, a) \text{ is true}\}$ (e.g., `PlayerMovementLaw` and `ZombieMovementLaw` in Fig. 2). The model assumes that all observables are *conditionally independent* given the current state and action. The predictive distribution for a single observable o is formed by combining the predictions from all active laws that have an opinion on it. Let $\mathcal{I}_o(s, a) = \{i \in \mathcal{I}(s, a) \mid o \in \mathcal{O}\}$ be the set of active laws relevant to observable o. The probability of observing an outcome v for this observable is given by a weighted-product of conditional probability distribution from each law, parameterized by $\boldsymbol{\theta}$:

$$p(o = v|s, a; \boldsymbol{\theta}) \propto \prod_{i \in \mathcal{I}_o(s,a)} \phi_i(o = v|s, a)^{\theta_i} \tag{1}$$

The complete predictive distribution over the next state s' is the product of the individual observable distributions:

$$p(s'|s, a; \boldsymbol{\theta}) = \prod_{o \in \mathcal{O}} p(o|s, a; \boldsymbol{\theta}) \tag{2}$$

3.3 ONELIFE: UNGUIDED ENVIRONMENT EXPLORATION AND LAW SYNTHESIS

The set of candidate laws L_i is generated from unguided agent-environment interactions through a two-stage process. First, an autonomous exploration policy gathers a corpus of interaction data. Second, a synthesizer proposes candidate laws that explain the state transitions observed in this data.

Exploration Policy. Previous work in symbolic world modeling often assumes access to curated offline datasets or utilizes online interaction guided by human-provided goals or environment rewards. In our unsupervised setting, such guidance is *unavailable*. Furthermore, in a hostile environment such as Crafter-OO, a simple random policy fails to survive long enough to experience the diverse mechanics necessary for comprehensive world modeling. Therefore, we employ an exploration policy driven by a large language model. The policy is not provided with specific knowledge of the environment; instead, it is given the high-level objective to discover as many underlying mechanics as possible, treating exploration as a reverse-engineering task. We use the agent scaffolding from Balrog (Paglieri et al., 2025) to implement the agent. The agent's architecture maintains a rolling window of its recent state-action history to provide context for decisions. The prompt (see Appendix F) also instructs the agent to maintain a transient summary of its current understanding of the world's rules, refining its hypotheses as it interacts with the environment.

Law Synthesizer. The synthesizer's task is to propose laws explaining the experienced transitions. Prior approaches (Piriyakulkij et al., 2025) have often relied on a large suite of hand-designed synthesizers, each tailored to specific types of interactions. This method embeds significant domain knowledge, which runs contrary to our goal of unsupervised discovery. We instead adopt a more general approach where a synthesizer is prompted to propose a large set of simple, *atomic* laws for each observed transition. An atomic law is one that describes a change to a minimal number of state attributes. For instance, a complex combat event involving the player and a zombie – resulting in changes to both entities' positions and health – is not modeled by a single monolithic law. Our synthesizer decomposes the event into multiple atomic laws: one for the player's health decrease, another for the zombie's movement, and so on. This decomposition into fine-grained, modular laws allows the subsequent weight-fitting stage Sec. 3.4 to perform more precise credit assignment, isolating and down-weighting incorrect hypotheses without discarding entire complex rules that may be partially correct. We provide examples of laws in Appendix A.

3.4 ONELIFE: INFERENCE ON LAW PARAMETERS

We learn the weight vector $\boldsymbol{\theta}$ by maximizing the log-likelihood of a dataset of observed transitions $\mathcal{D} = \{(s_t, a_t, s_{t+1})\}_{t=1}^{N}$. For clarity, we first define the loss for a single transition (s, a, s'); the total loss is the sum over all transitions in the dataset.

Based on the conditional independence of observables, the negative log-likelihood for a single transition decomposes into a sum over each observable $o \in \mathcal{O}$:

$$\mathcal{L}(\boldsymbol{\theta}; s, a, s') = -\sum_{o \in \mathcal{O}} \log p(v_o^* | s, a; \boldsymbol{\theta}) \tag{3}$$

where $v_o^* = \mathcal{E}(s')_o$ is the ground truth value of observable o extracted from the next state s'. The log-probability term is derived from the combined predictions of the active laws. Let $\mathcal{I}_o(s, a)$ be the set of active laws that make a prediction for observable o. We first define the combined, unnormalized log-score for any potential value v as the weighted sum of log-scores from these laws. The weights θ_i are the *only learnable parameters*:

$$\ell_o(v | s, a; \boldsymbol{\theta}) = \sum_{i \in \mathcal{I}_o(s,a)} \theta_i \cdot \phi_{i,o}(v | s, a) \tag{4}$$

Normalized log-probability of observing the specific outcome v_o^* is then given by the log-softmax function. Let supp(o) be the discrete support (set of all possible values) for observable o:

$$\log p(v_o^* | s, a; \boldsymbol{\theta}) = \ell_o(v_o^* | s, a; \boldsymbol{\theta}) - \log \sum_{v \in \text{supp}(o)} \exp\left(\ell_o(v | s, a; \boldsymbol{\theta})\right) \tag{5}$$

The optimization process leverages the dynamic computation graph induced by our law structure. For each transition and each observable, the loss gradient is calculated with respect to the weights

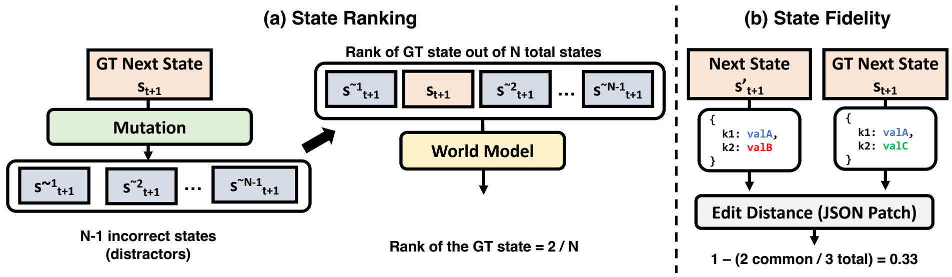

Figure 3: Two evaluation metric categories described in Sec. 4. A world state of an environment usually has more than two keys (*i.e. Crafter-OO's state (Listing 1) when populated has 100+ key-value pairs,*) and often has nested values, but here we show a simplest case to explain the calculation of (normalized) edit distance. We create distractors for state ranking using *mutators* (Appendix C), which programatically modify the next state s' in a transition (s, a, s') to be illegal under the true transition function. For example, one of our mutators allows a crafting action (e.g. making a stone pickaxe) to succeed even when the prequisites for the crafting are not met.

θ_i only for the active laws $i \in \mathcal{I}_o(s_t, a_t)$. This effectively **routes** credit for an outcome exclusively to the laws that made a prediction about it. This sparse, targeted update mechanism provides more precise credit assignment than methods that update a global set of weights based on aggregate outcomes. We use L-BFGS for optimization (Nocedal & Wright, 2006).

3.5 OneLife: Forward Simulation and Likelihood

Forward simulation is the process of using the learned world model generatively to predict a future state $\hat{s}_{t+1}$ given a current state s_t and an action a_t. By generating rollouts of future trajectories, an agent can evaluate action sequences against a specific goal or reward function without costly or irreversible real-world interaction.

The simulation of a single timestep from (s_t, a_t) involves a multi-step sampling and reconstruction process. First, for each observable $o \in \mathcal{O}$, the model forms a predictive probability distribution $p(o|s_t, a_t; \boldsymbol{\theta})$. This distribution is constructed by identifying the set of active laws $\mathcal{I}_o(s_t, a_t)$ relevant to that observable and combining their predictions according to their learned weights θ_i, as specified in Equation 1. This distribution can be used to evaluate the likelihood of an observable conditioned on (s, a) pair. Second, a concrete outcome $\hat{v}_o$ can be sampled from this distribution for each observable: $\hat{v}_o \sim p(o|s_t, a_t; \boldsymbol{\theta})$. This the collection of sampled outcomes $\{\hat{v}_o\}_{o \in \mathcal{O}}$ is used to construct the full symbolic next state $\hat{s}_{t+1}$. A reconstruction function, which mirrors the observable extraction process, assembles these values back into the environment's structured state representation.

4 Evaluation Protocols and Metrics

The evaluation of world models for a stochastic environment is non-trivial. An useful world model fulfills two criteria: (a) **state ranking**, the ability to distinguish plausible future states from implausible ones, and (b) **state fidelity**, the ability to generate future states that closely resemble reality. Both are illustrated in Fig. 3.

State Ranking (Fig. 3 (a)). These metrics assess the model's ability to rank the true next state higher than the distractors. To create the distractor states, we use **mutators**, which are programmatic functions that apply semantically meaningful, rule-breaking changes to the true next state. For example, a mutator could change a character's position to a location they cannot physically reach. We include details on mutators in Appendix C.

- **Rank @ 1 (R@1):** A binary metric that measures whether the model correctly assigns the highest probability (rank 1) to the true next state among all candidates.
- **Mean Reciprocal Rank (MRR):** This metric averages the reciprocal rank of the correct answer across all test instances. A higher MRR indicates that the model consistently ranks the correct state higher. The formula is: MRR $= \frac{1}{N}\sum_{i=1}^{N}\frac{1}{r_i}$, where r_i is the rank of the ground truth state for the i-th transition, with rank 1 being the highest probability.

Table 1: Performance comparison of world modeling methods on the Crafter-OO environment, averaged over ten trials. We evaluate models on two criteria: **state fidelity** and **state ranking** All methods use the OneLife exploration policy and law synthesizer but differ in their parameter inference method. OneLife shows significant improvements over the PoE-World inference algorithm and OneLife variant without parameter inference. The random baseline is shaded in gray.

Law Synthesis (Sec. 3.3)	Law Param. Inference (Sec. 3.4)	State Ranking: Rank @ 1 ↑	State Ranking: MRR ↑	State Fidelity: Raw Edit Dist. ↓	State Fidelity: Norm. Edit Dist. ↓
Random World Model		8.5%	0.322	121.538	0.809
OneLife	PoE-World	10.8%	0.351	10.634	0.071
OneLife	None	13.0%	0.429	8.540	0.057
OneLife	OneLife	18.7%	0.479	8.764	0.058
Δ over PoE-World		(+7.9%)	(+0.128)	(-1.870)	(-0.013)

State Fidelity (Fig. 3 (b)). These measure the error between predicted and ground truth states.

- **Raw Edit Distance:** The total number of atomic JSON Patch operations required to transform the predicted state, s'_{t+1}, into the ground truth state, s_{t+1}.
- **Normalized Edit Distance:** The raw edit distance divided by the total number of elements in the state representation.

5 Experimental Setup and Results

5.1 Evaluation Framework Implementation on Crafter-OO

Evaluating a world model on random rollouts may not provide sufficient coverage of rare or important events in an environment. To ensure our evaluation is comprehensive, we create evaluation trajectories from a suite of **scenarios**. Each scenario runs short, scripted policy from an initial state designed to reliably exercise a specific game mechanic or achieve a particular goal, ensuring that our evaluation thoroughly covers the environment's dynamics. Our scenarios cover every achievement in the achievement tree of Crafter-OO/Crafter, and can be seen in Fig. 4. We generate a comprehensive evaluation dataset by implementing scenarios that cover every achievement in the game's achievement tree. This ranges from basic actions like collecting wood to complex, multi-step tasks like crafting an iron sword, ensuring all of the game's core mechanics are tested. More details on scenarios are provided in Appendix D. We generate distractors for each transition in the evaluation dataset using a bank of 8 mutators which each produce a subtle, but illegal transformation of the game state in response to an action. Some examples are causing an incorrect item to be produced when taking a crafting action, or allowing an item to be produced without the correct requirements, or illegal entity behavior such as teleporting. More details on mutators are provided in Appendix C and general implementation details are in Appendix E.

5.2 Baseline Models

To contextualize the results of our proposed model, we compare against several two baselines:

- **Random World Model:** A model that assigns a uniform probability to all candidate states in the discriminative task. Its performance is equivalent to random guessing and serves as a sanity check for discriminative accuracy.
- **PoE-World** (Piriyakulkij et al., 2025): A state-of-the-art symbolic world model that scaled symbolic world modeling to domains like Atari. Both PoE-World and OneLife represent the transition function as a product of programs, though the structure of the programs is different. Because PoE-World's law synthesis component is Atari-specific and relies on online interaction using human-provided goals, we reimplement this baseline with our exploration policy and law synthesizer, noting that this makes it a stronger baseline (without these changes, PoE-World's Atari-specific nature means it would fail on the vast majority of transitions).

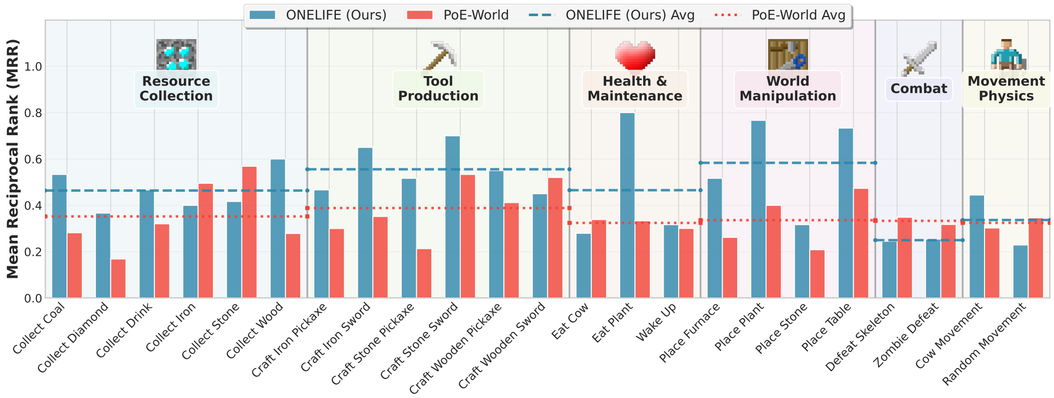

Figure 4: Per-scenario state ranking performance of ONELIFE versus PoE-World, measured by Mean Reciprocal Rank (MRR ↑). Scenarios are grouped by the core game mechanic they test. Horizontal lines show the average MRR across all scenarios in a group for ONELIFE and PoE-World. ONELIFE demonstrates a more accurate understanding of the environment's laws, achieving a higher average MRR and outperforming the baseline on the majority of individual scenarios.

5.3 RESULTS

State Fidelity and Ranking. ONELIFE learns a world model with significantly higher predictive judgment than baseline methods while maintaining competitive generative fidelity. Table 1 compares our full method against baselines and key ablations across all evaluation metrics. ONELIFE's primary advantage appears in the predictive judgment metrics. We achieve a discriminative accuracy of 18.7% and an MRR of 0.479, outperforming the PoE-World optimization baseline by 7.9 percentage points and 0.128, respectively. While precisely generating a complex future state remains challenging, our model has learned an accurate understanding of the environment's underlying laws. This enables it to assign high probability to valid transitions and low probability to invalid ones. The comparison to the "random world model" shows that (i) a high edit distance can quickly be amassed if the world models updates observables that are unchanged in the ground truth state, thus, reinforcing why such simulation is challenging. (ii) Optimizing for generative metrics like state fidelity alone *does not* yield a better world model to guide an agent, e.g., while the PoE-world model (row 2 in Tab. 1) dramatically improves the state fidelity by reducing the edit distance *a factor of 10*, it only *marginally* improves the ability to *rank multiple states* by ≈2% over random (Rank@1) – reiterating the need for state ranking metrics.

Fine-grained Evaluation. Figure 4 breaks down Mean Reciprocal Rank performance across individual scenarios spanning mechanics from resource collection to combat. ONELIFE consistently outperforms the PoE-World baseline on the majority (16/23) of scenarios. These improvements stem from a robust understanding of the environment's diverse rules rather than strong performance on only a few simple mechanics.

6 CONCLUSION

We address the problem of learning a symbolic world model from limited, unguided interaction in a complex, stochastic environment. We introduced ONELIFE, a framework that represents world dynamics as a probabilistic mixture of modular, programmatic laws. Its core learning mechanism routes credit for observed state changes exclusively to the laws responsible for predicting them, enabling effective learning even when many rules are inactive during a given transition. Evaluated on Crafter-OO, our variant of the complex Crafter environment with object-centric state, ONELIFE learns a world model with superior predictive judgment compared to a strong baseline, more accurately distinguishing plausible future states from implausible ones. This improvement is consistent across a wide range of game mechanics. Our work provides a foundation for building agents that can autonomously reverse engineer the rules of an unknown environment.

ETHICS STATEMENT

We do not foresee any ethical implications beyond standard ethical and safety considerations that apply to AI research generally.

REPRODUCIBILITY STATEMENT

We plan to open-source Crafter-OO, ONELIFE, and the evaluation framework used in our work to aid reproducibility. All prompts and key details of the exploration policy, synthesis algorithm, and law parameter inference have been described.

REFERENCES

Zergham Ahmed, Joshua B Tenenbaum, Christopher J Bates, and Samuel J Gershman. Synthesizing world models for bilevel planning. *TMLR*, 2025.

P. (Ed.) Bryan and M. (Ed.) Nottingham. JavaScript Object Notation (JSON) Patch. RFC 6902, IETF, April 2013. URL https://datatracker.ietf.org/doc/html/rfc6902.

Yimeng Chen, Piotr Piekos, Mateusz Ostaszewski, Firas Laakom, and Juergen Schmidhuber. Physgym: Benchmarking llms in interactive physics discovery with controlled priors. *arXiv preprint*, 2025.

Maxime Chevalier-Boisvert, Dzmitry Bahdanau, Salem Lahlou, Lucas Willems, Chitwan Saharia, Thien Huu Nguyen, and Yoshua Bengio. Babyai: A platform to study the sample efficiency of grounded language learning. *arXiv preprint arXiv:1810.08272*, 2018.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo Perez-Vicente, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. In *Advances in Neural Information Processing Systems 36, New Orleans, LA, USA*, December 2023.

Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. Generating code world models with large language models guided by monte carlo tree search. In *NeurIPS*, 2024.

Ria Das, Joshua B. Tenenbaum, Armando Solar-Lezama, and Zenna Tavares. Combining functional and automata synthesis to discover causal reactive programs. *Proc. ACM Program. Lang.*, 7 (POPL), January 2023. doi: 10.1145/3571249. URL https://doi.org/10.1145/3571249.

Antoine Dedieu, Joseph Ortiz, Xinghua Lou, Carter Wendelken, Wolfgang Lehrach, J Swaroop Guntupalli, Miguel Lazaro-Gredilla, and Kevin Patrick Murphy. Improving transformer world models for data-efficient rl. In *ICML*, 2025.

Quentin Delfosse, Jannis Blüml, Bjarne Gregori, Sebastian Sztwiertnia, and Kristian Kersting. OCAtari: Object-centric Atari 2600 reinforcement learning environments. *Reinforcement Learning Journal*, 1:400–449, 2024.

Boyuan Deng, Qiaochu Xu, Hao Li, Ling Li, Rui Xu, Yunlong Lu, Jie Lin, and Ziqi Zhang. Pddlego: Iterative planning in textual environments. *arXiv preprint*, 2024.

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021. URL http://github.com/google/brax.

Jiayi Geng, Howard Chen, Dilip Arumugam, and Thomas L Griffiths. Are large language models reliable ai scientists? assessing reverse-engineering of black-box systems. *arXiv preprint*, 2025.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pretrained large language models to construct and utilize world models for model-based task planning. In *NeurIPS*, 2023.

Danijar Hafner. Benchmarking the spectrum of agent capabilities. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=1W0z96MFEoH.

Danijar Hafner, Jurgis Pašukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint*, 2023.

Peter Jansen, Marc-Alexandre Côté, Tushar Khot, Erin Bransom, Bhavana Dalvi Mishra, Bodhisattwa Prasad Majumder, Oyvind Tafjord, and Peter Clark. Discoveryworld: A virtual environment for developing and evaluating automated scientific discovery agents. In *NeurIPS*, 2024.

Martin Klissarov, Mikael Henaff, Roberta Raileanu, Shagun Sodhani, Pascal Vincent, Amy Zhang, Pierre-Luc Bacon, Doina Precup, Marlos C Machado, and Pierluca D'Oro. Maestromotif: Skill design from artificial intelligence feedback. In *ICLR*, 2025.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. *arXiv preprint*, 2022.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *ICLR*, 2024.

Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Jackson, Samuel Coward, and Jakob Foerster. Craftax: A lightning-fast benchmark for open-ended reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2024.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – the planning domain definition language. Technical report, Yale Center for Computational Vision and Control, 1998.

Vincent Micheli, Eloi Alonso, and François Fleuret. Transformers are sample-efficient world models. In *ICLR*, 2023. Oral Presentation.

Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, second edition, 2006.

Davide Paglieri, Bartłomiej Cupiał, Samuel Coward, Ulyana Piterbarg, Maciej Wolczyk, Akbir Khan, Eduardo Pignatelli, Łukasz Kuciński, Lerrel Pinto, Rob Fergus, et al. Balrog: Benchmarking agentic llm and vlm reasoning on games. In *The Thirteenth International Conference on Learning Representations*, 2025.

Wasu Top Piriyakulkij, Yichao Liang, Hao Tang, Adrian Weller, Marta Kryven, and Kevin Ellis. Poe-world: Compositional world modeling with products of programmatic experts. *arXiv preprint*, 2025.

Max Schwarzer, Nitarshan Rajkumar, Michael Noukhovitch, Ankesh Anand, Laurent Charlin, Devon Hjelm, Philip Bachman, and Aaron Courville. Pretraining representations for data-efficient reinforcement learning. In *NeurIPS*, 2021.

Ramanan Sekar, Oleh Rybkin, Kostas Daniilidis, Pieter Abbeel, Danijar Hafner, and Deepak Pathak. Planning to explore via self-supervised world models. In *ICML*, 2020.

Elias Stengel-Eskin, Archiki Prasad, and Mohit Bansal. Regal: Refactoring programs to discover generalizable abstractions. In *International Conference on Machine Learning*, pp. 46605–46624. PMLR, 2024.

Hao Tang, Darren Key, and Kevin Ellis. Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment. *arXiv preprint*, 2024.

Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. Learning to synthesize programs as interpretable and generalizable policies. In *NeurIPS*, 2021.

Keyon Vafa, Joshua B Tenenbaum, Jacob D Andreas, Yoshua Bengio, and Brenden M Lake. Evaluating the world model implicit in a generative model. In *NeurIPS*, 2024.

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming, 2021. URL https://arxiv.org/abs/1809.10756.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *TMLR*, 2025.

Lance Ying, Katherine M Collins, Prafull Sharma, Cedric Colas, Kaiya Ivy Zhao, Adrian Weller, Zenna Tavares, Phillip Isola, Samuel J Gershman, Jacob D Andreas, Thomas L Griffiths, Francois Chollet, Kelsey R Allen, and Joshua B Tenenbaum. Assessing adaptive world models in machines with novel games. In *ICLR*, 2025.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, et al. Language to rewards for robotic skill synthesis. In *CoRL*, 2023.

A Law Examples

Below, we give examples of various laws synthesized by OneLife. In box A.1 and box A.2, we show examples of how OneLife has learned the hierarchical structure of Crafter-OO/Crafter's tech-tree. In this case, one must mine stone before a stone pickaxe can be produced. These laws are deterministic. In box A.3, we give an example of a law synthesized by OneLife for a stochastic mechanic, in this case, the chase behavior of zombies when they are within a certain range of a player. The idle skeleton law in box A.6 and moving skeleton law in box A.4 make conflicting predictions; these are aggregated by the weight inference process in Sec. 3.4 to produce a distributional prediction that takes into account the predictive accuracy of both laws.

Box A.1| Mine Stone Law

```python
class MineStoneLaw:
    def __init__(self):
        """Initialize with configurable parameters."""
        pass

    def precondition(self, current_state: WorldState, action: str) -> bool:
        """Return True if this law should apply to the given state and action."""
        if action != "Do":
            return False

        target_material, _ = current_state.get_target_tile()

        if target_material == "stone":
            # Check if the player has any pickaxe
            has_pickaxe = (
                current_state.player.inventory.wood_pickaxe > 0 or
                current_state.player.inventory.stone_pickaxe > 0 or
                current_state.player.inventory.iron_pickaxe > 0
            )
            return has_pickaxe

        return False

    def effect(self, current_state: WorldState, action: str) -> None:
        """Apply the law by modifying the world state."""
        # Increment stone in inventory
        current_state.player.inventory.stone = DiscreteDistribution(
            support=[current_state.player.inventory.stone + 1]
        )

        # Replace the mined stone material with grass
        current_state.set_facing_material("grass")
```

Box A.2| Craft Stone Pickaxe

```
class CraftStonePickaxe:
    def __init__(self):
        """Initialize with configurable parameters."""
        # No specific parameters needed for this crafting recipe.
        pass

    def precondition(self, current_state: WorldState, action: str) -> bool:
        """Return True if this law should apply to the given state and action."""
        # Check if the action is "Make Stone Pickaxe"
        if action == "Make Stone Pickaxe":
            # Check if player has required materials
            has_wood = current_state.player.inventory.wood >= 1
            has_stone = current_state.player.inventory.stone >= 1
            return has_wood and has_stone
        return False

    def effect(self, current_state: WorldState, action: str) -> None:
        """Apply the law by modifying the world state."""
        # Decrease wood by 1
        current_state.player.inventory.wood = DiscreteDistribution(support=[
    current_state.player.inventory.wood - 1])
        # Decrease stone by 1
        current_state.player.inventory.stone = DiscreteDistribution(support=[
    current_state.player.inventory.stone - 1])
        # Increase stone_pickaxe by 1
        current_state.player.inventory.stone_pickaxe = DiscreteDistribution(
    support=[current_state.player.inventory.stone_pickaxe + 1])
```

Box A.3| Zombie Chase

```
class ZombieAggroMovement:
    def __init__(self):
        """Initialize with configurable parameters."""
        pass # No specific parameters are needed for this observed law.

    def precondition(self, current_state: WorldState, action: str) -> bool:
        """Return True if this law should apply to the given state and action."""
        # This law applies if there are any ZombieState entities within the
    player's
        # update range, as their movement is an autonomous process.
        zombies_in_range = current_state.get_object_of_type_in_update_range(
    ZombieState)
        return len(zombies_in_range) > 0

    def effect(self, current_state: WorldState, action: str) -> None:
        """Apply the law by modifying the world state."""
        player_pos = current_state.player.position

        # Retrieve all ZombieState objects that are within the update range.
        # This implicitly filters for zombies close enough to be active/
    observable.
        zombies_to_update = current_state.get_object_of_type_in_update_range(
    ZombieState)

        for zombie in zombies_to_update:
            # Calculate the differences in coordinates between the player and the
     zombie.
            dx = player_pos.x - zombie.position.x
```

```
            dy = player_pos.y - zombie.position.y

            # Initialize new positions to current positions (no movement by
        default)
            new_x = zombie.position.x
            new_y = zombie.position.y

            # Prioritize movement along the X-axis
            if dx != 0:
                # Move one step towards the player along the X-axis.
                new_x = zombie.position.x + (1 if dx > 0 else -1)
            elif dy != 0:
                # If X-axis is already aligned, move one step towards the player
        along the Y-axis.
                new_y = zombie.position.y + (1 if dy > 0 else -1)

            # Update the zombie's position in the state using
        DiscreteDistribution.
            zombie.position.x = DiscreteDistribution(support=[new_x])
            zombie.position.y = DiscreteDistribution(support=[new_y])
```

Box A.4| Skeleton Movement

```
class SkeletonRandomMovementLaw:
    def __init__(self):
        """Initialize with configurable parameters."""
        pass

    def precondition(self, current_state: WorldState, action: str) -> bool:
        """Return True if this law should apply to the given state and action."""
        # This law applies generally to all skeletons, independent of player
    action for movement
        return True

    def effect(self, current_state: WorldState, action: str) -> None:
        """Apply the law by modifying the world state."""
        skeletons = [obj for obj in current_state.objects if isinstance(obj,
    SkeletonState)]

        for skeleton in skeletons:
            current_x = skeleton.position.x
            current_y = skeleton.position.y

            # Possible next X positions: current_x, current_x + 1, current_x - 1
            skeleton.position.x = DiscreteDistribution(support=[
                current_x,
                current_x + 1,
                current_x - 1
            ])
            # Possible next Y positions: current_y, current_y + 1, current_y - 1
            skeleton.position.y = DiscreteDistribution(support=[
                current_y,
                current_y + 1,
                current_y - 1
            ])
```

Box A.5| Health Regeneration Law

```
class PlayerInventoryHealthRegeneration:
    def __init__(self, max_health: int = 20, recover_threshold: float = 1.0):
        """Initialize with configurable parameters for health regeneration."""
        self.max_health = max_health
        self.recover_threshold = recover_threshold

    def precondition(self, current_state: WorldState, action: str) -> bool:
        """
        Return True if the player's inventory health should regenerate.
        This law applies if the player is not at max health, has sufficient
        recover points, and is not sleeping.
        """
        player = current_state.player

        # Check if player's current inventory health is less than the defined
        maximum
        has_space_for_health = player.inventory.health < self.max_health

        # Check if player has sufficient recover points to enable regeneration
        has_recover_points = player.recover >= self.recover_threshold

        # Check if the player is not currently sleeping
        not_sleeping = not player.sleeping

        # This is a passive regeneration effect, so the specific action taken (e.
        g., "Move North")
        # is not a direct precondition, but the effect occurs during the state
        transition.
        return has_space_for_health and has_recover_points and not_sleeping

    def effect(self, current_state: WorldState, action: str) -> None:
        """
        Apply the law by increasing the player's inventory health by 1.
        """
        # Increment the player's inventory health by 1.
        current_state.player.inventory.health = DiscreteDistribution(support=[
        current_state.player.inventory.health + 1])
```

Box A.6| Skeleton Idle

```
class SkeletonIdleLaw:
    def __init__(self):
        """Initialize with configurable parameters."""
        pass

    def precondition(self, current_state: WorldState, action: str) -> bool:
        """Return True if this law should apply to the given state and action."""
        # This law applies if there are any skeletons in the world that aren't
        otherwise engaged.
        # Since no changes were observed, we assume this is their default passive
         behavior.
        return True # Applies universally as a default behavior for skeletons

    def effect(self, current_state: WorldState, action: str) -> None:
        """Apply the law by modifying the world state."""
        for skeleton in current_state.get_object_of_type_in_update_range(
        SkeletonState):
            # Based on observation, skeletons remain unchanged.
```

```
            # We predict their attributes will stay the same.
            skeleton.health = DiscreteDistribution(support=[skeleton.health])
            skeleton.position.x = DiscreteDistribution(support=[skeleton.position
    .x])
            skeleton.position.y = DiscreteDistribution(support=[skeleton.position
    .y])
            skeleton.reload = DiscreteDistribution(support=[skeleton.reload])
```

B The Crafter-OO Environment

This appendix details Crafter-OO, our reimplementation of the Crafter environment that exposes a structured, object-oriented symbolic state and operates through a pure transition function. We developed Crafter-OO as a testbed for symbolic world modeling approaches in a complex, stochastic domain.

B.1 Motivation and Design Principles

Symbolic world modeling benefits from environments where the complete state is accessible as a structured representation. Simple grid worlds provide this but lack complexity, while more complex environments typically require additional engineering to expose their internal state. More fundamentally, existing testbeds for symbolic world modeling have focused on environments that are either deterministic or have limited stochasticity and a narrow range of mechanics. Atari games, for instance, while complex in visual processing demands, have relatively predictable dynamics and a constrained set of interactions compared to open-world environments.

We developed Crafter-OO to address this gap. The environment features significant stochasticity in entity behaviors, diverse mechanics spanning resource collection to combat, and multi-step causal chains. Our design follows three principles:

1. **Explicit Object-Oriented State**: The entire game state is captured in a single, hierarchical data model that serves as input and output for world models.
2. **Functional Purity**: The environment's dynamics are exposed as a pure transition function, $T(\text{state}, \text{action}) \rightarrow \text{next\_state}$, with no hidden variables.
3. **Programmatic Modification**: The state representation can be precisely manipulated with code, enabling controlled experimental setups.

B.2 The `WorldState` Data Model

The core of Crafter-OO is the `WorldState` data model, which captures the environment at a single timestep. This model is defined using Pydantic for structure and validation. Its components include:

- `player`: A `PlayerState` object containing position, inventory, health, and current action.
- `objects`: A list of non-player entities (`CowState`, `ZombieState`, `PlantState`, etc.) with type discrimination via a `name` field.
- `materials`: A 2D array representing the terrain map.
- **Global Properties**: World-level attributes including `daylight`, `size`, and serialized random state.

Listing 1 shows the structure of this model. This representation provides the interface between the environment and symbolic world models.

```
from typing import TypeAlias, Literal

# --- Basic Data Structures ---

class Position:
    """Represents a 2D position (x, y) in the game world."""
    x: int
    y: int

```

```python
class Inventory:
    """Represents the player's inventory counts for each item type."""
    health: int
    food: int
    drink: int
    energy: int
    sapling: int
    wood: int
    stone: int
    coal: int
    iron: int
    diamond: int
    wood_pickaxe: int
    stone_pickaxe: int
    iron_pickaxe: int
    wood_sword: int
    stone_sword: int
    iron_sword: int

class Achievements:
    """Represents the player's unlocked achievements."""
    collect_coal: int
    collect_diamond: int
    collect_drink: int
    collect_iron: int
    collect_sapling: int
    collect_stone: int
    collect_wood: int
    defeat_skeleton: int
    defeat_zombie: int
    eat_cow: int
    eat_plant: int
    make_iron_pickaxe: int
    make_iron_sword: int
    make_stone_pickaxe: int
    make_stone_sword: int
    make_wood_pickaxe: int
    make_wood_sword: int
    place_furnace: int
    place_plant: int
    place_stone: int
    place_table: int
    wake_up: int


# --- Game World Entities ---

class BaseObject:
    """The base class for all dynamic objects in the game world."""
    entity_id: int
    position: Position
    health: int
    removed: bool

class Player(BaseObject):
    """The state of the player character."""
    name: Literal["player"] = "player"
    facing: Position
    action: str
    sleeping: bool
    inventory: Inventory
    achievements: Achievements
    thirst: float
    hunger: float
    fatigue: float
```

```python
    recover: float
    last_health: int

class Cow(BaseObject):
    """The state of a cow."""
    name: Literal["cow"] = "cow"

class Zombie(BaseObject):
    """The state of a zombie."""
    name: Literal["zombie"] = "zombie"
    cooldown: int

class Skeleton(BaseObject):
    """The state of a skeleton."""
    name: Literal["skeleton"] = "skeleton"
    reload: int

class Arrow(BaseObject):
    """The state of an arrow projectile."""
    name: Literal["arrow"] = "arrow"
    facing: Position

class Plant(BaseObject):
    """The state of a plant, which can be eaten."""
    name: Literal["plant"] = "plant"
    grown: int
    ripe: bool

class Fence(BaseObject):
    """The state of a fence object."""
    name: Literal["fence"] = "fence"

# A union of all possible entity types in the world.
Entity: TypeAlias = Player | Cow | Zombie | Skeleton | Arrow | Plant | Fence


# --- World and Spatial Structures ---

MaterialT: TypeAlias = str

class Chunk:
    """Represents a spatial region of the world for efficient updates."""
    chunk_key: tuple[int, int, int, int]
    object_ids: list[int]

class WorldState:
    """Represents the complete, hierarchical state of the game world at a single
     timestep."""
    # World dimensions and configuration
    size: tuple[int, int]
    chunk_size: tuple[int, int]
    view: tuple[int, int]

    # World status
    daylight: float
    step_count: int

    # The grid of static materials (e.g., grass, stone, water)
    materials: list[list[MaterialT | None]]

    # A list of all dynamic entities currently in the world.
    objects: list[Entity]

    # A direct reference to the player object for easy access.
    player: Player
```

```
    # Spatial partitioning data.
    chunks: list[Chunk]

    # Internal simulation state
    entity_id_counter_state: int
    serialized_random_state: str
    event_bus: list[str]
```

Listing 1: Simplified structure of the `WorldState` data structure.

B.3 Extracting State from Crafter's Game Engine

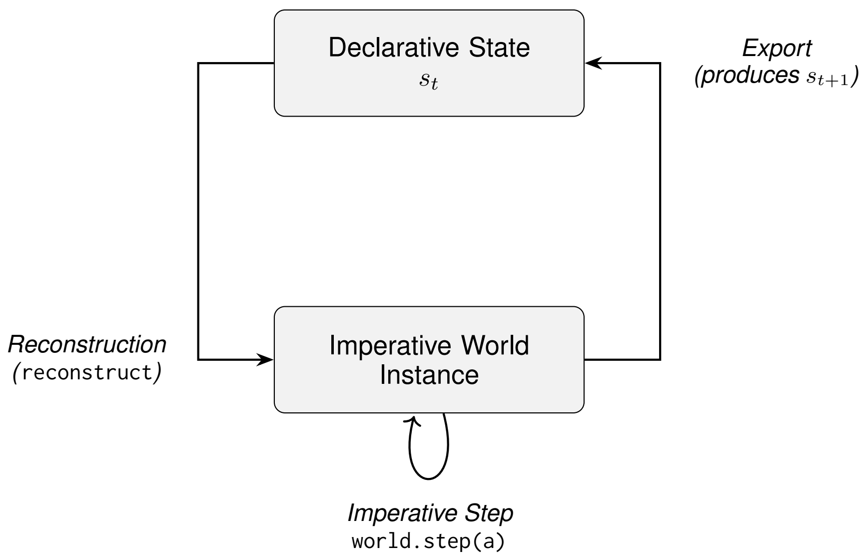

Figure 5: The functional cycle for state transition. A declarative state snapshot is reconstructed into a live, imperative world instance. The engine simulates a single step, and the resulting world is exported back into a new declarative state snapshot for the next timestep. This ensures we match Crafter's mechanics exactly.

The simulation state in the original engine is not a single data structure but is distributed across a graph of live Python objects, each with its own internal state and complex inter-dependencies, such as non-player characters holding direct references to the player object. Furthermore, the engine's behavior relies on implicit state, including the internal state of its pseudo-random number generator, which governs all stochastic events. Achieving a pure functional interface required developing a robust mechanism to first serialize this entire, complex state into a self-contained, declarative representation and then perfectly reconstruct the live object graph from that representation for each step of the simulation.

The state export process transforms the live simulation into a serializable snapshot. This procedure performs a deep traversal of the game engine's internal state, capturing all information required to reproduce the exact game moment. This includes the grid of world materials, the positions of all entities, and the type-specific attributes of each entity, such as a zombie's attack cooldown or a plant's growth progress. Crucially, the process also serializes the state of the engine's pseudo-random number generator, ensuring that the sequence of random numbers for subsequent stochastic events is preserved. To maintain the spatial partitioning data used for efficient queries, the set of entities within each world chunk is recorded by storing their unique identifiers. The final output is a complete, declarative data structure that represents the world at a single point in time, free from any live object references or other runtime-specific information.

State reconstruction reverses this process, rebuilding the live simulation from the declarative snapshot. This is more complex than simply loading data. It involves re-instantiating the entire graph of game objects and correctly re-establishing their inter-dependencies. A key complexity arises from object relationships; for instance, hostile entities require a direct reference to the live player object to guide their behavior. To resolve this, we employ a multi-pass reconstruction algorithm. First, entities with no external dependencies, such as the player, are instantiated. Then, dependent entities are instantiated in a second pass, receiving references to the already-created objects they require. Once all objects are created, the spatial partitioning system is rebuilt by mapping the stored entity identifiers back to the newly created live object instances. Finally, the deserialized state of the pseudo-random number generator is loaded, ensuring that the reconstructed world will produce the exact same stochastic outcomes as the original. The overall process is described in Box 1 and illustrated in Figure 1.

Box B.1| Pseudocode for the Functional Transition Cycle

```
function FunctionalTransition(declarative_state_t, action_t):
 // 1. Reconstruct the imperative world from the declarative state snapshot.
  world_instance <- ReconstructWorldFromState(declarative_state_t)

  // 2. Emulate a single step in the imperative engine.
  player <- FindPlayerObject(world_instance)
  ApplyActionToPlayer(player, action_t)
  for object in world_instance.get_all_objects():
    object.update()

  // 3. Export the new world state into a declarative representation.
  declarative_state_t+1 <- ExportStateFromWorld(world_instance)

  return declarative_state_t+1

function ExportStateFromWorld(world_instance):
  snapshot <- new DeclarativeState
  snapshot.materials <- CopyGrid(world_instance.material_grid)
  snapshot.rng_state <- Serialize(world_instance.random_generator)
  for object in world_instance.get_all_objects():
    AddObjectState(snapshot, object.type, object.attributes, object.id)
  return snapshot

function ReconstructWorldFromState(snapshot):
  world_instance <- new ImperativeWorld
  world_instance.material_grid <- CopyGrid(snapshot.materials)
  world_instance.random_generator <- Deserialize(snapshot.rng_state)

  // Multi-pass object instantiation to handle dependencies.
  player_state <- FindPlayerStateInSnapshot(snapshot)
  player_object <- InstantiateObject(
    player_state.type, player_state.attributes
  )
  AddObjectToWorld(world_instance, player_object)

  for object_state in snapshot.get_all_object_states():
    if not is_player(object_state):
      // Pass player reference to dependent objects (e.g., Zombie).
      dependencies <- {player: player_object}
      new_object <- InstantiateObject(
        object_state.type, object_state.attributes, dependencies
      )
      AddObjectToWorld(world_instance, new_object)

  RebuildSpatialIndex(world_instance)
```

```
    return world_instance
```

B.4 The Functional Environment Interface

We provide a `transition` function that implements a stateless API for environment steps:

1. Input: `WorldState` object s_t
2. Reconstruct live game engine instance
3. Execute single update tick with given action
4. Export resulting state as s_{t+1}
5. Return new `WorldState` object

This ensures every transition is a pure function of the explicit state, making the environment suitable for symbolic reasoning and program synthesis.

B.5 Utilities for Programmatic State Interaction

A key contribution of Crafter-OO is a rich set of utilities that enable programmatic interaction with the world state. These functions are essential for two purposes: first, they allow for the precise, reproducible setup of the evaluation scenarios discussed in Appendix D; second, they provide a high-level API that simplifies the authoring of programmatic world model laws. To provide a clear overview of this toolkit, Table 2 catalogues the key functions, which are grouped into three main categories: World Setup, Player State, and High-Level State Queries & Modifications.

Table 2: A catalogue of key utilities for programmatic state manipulation in Crafter-OO. These functions provide the building blocks for creating controlled experimental scenarios and for writing concise, high-level world model laws.

Category	Function Signature (Simplified)	Description
World Setup Utilities	`set_tile_material(pos, material)`	Modifies the terrain at a specific coordinate (e.g., changes `grass` to `stone`).
	`add_object_to_world(cls, pos, ...)`	Adds an entity instance (e.g., a `Cow` or `Zombie`) to the world.
	`remove_object_from_world(obj)`	Removes a specific entity instance from the world.
	`set_daylight(level)`	Sets the global daylight level, affecting visibility and mob spawning.
Player State Utilities	`set_player_position(pos)`	Sets the player's exact (x, y) coordinates.
	`set_player_facing(direction)`	Sets the player's facing direction (e.g., up, down, left, right).
	`set_player_inventory_item(item, qty)`	Sets the quantity of a specific item in the player's inventory.
	`set_player_internal_stat(stat, val)`	Adjusts internal player stats like health, hunger, or energy.
High-Level State Queries & Modifications	`get_target_tile()`	Returns the material and any object at the tile the player is facing.
	`get_object_of_type_in_update_range(cls)`	Returns all entities of a specific type near the player.
	`move_object(obj, dir, walkable)`	Moves an entity one step if the target tile is valid and unoccupied.
	`set_facing_material(material)`	Changes the material of the tile the player is facing.

These utilities are composed to construct the specific initial conditions for our evaluation scenarios. Listing 2 demonstrates how they work in concert to create a test case for a resource collection mechanic. World setup utilities are first used to clear an area and place a specific resource (`coal`). Then, player state utilities are used to position the player correctly and provide the necessary tool (`wood_pickaxe`) in their inventory. This level of programmatic control, enabled by the functions detailed in Table 2, is what makes our targeted evaluation methodology possible.

```python
def get_initial_state_for_coal_collection():
    # Create a base world and get references to the world and player objects
    world = reconstruct_world_from_state(initial_state())
    player = find_player(world)

    # --- World Setup Utilities ---
    # Clear a 3x3 area around the player to be grass
    for x in range(4, 7):
        for y in range(4, 7):
            world_utils.set_tile_material(world, (x, y), "grass")

    # Place the target resource in a specific location
    world_utils.set_tile_material(world, (6, 5), "coal")

```

```python
    # --- Player State Utilities ---
    # Set the player's starting position
    player_utils.set_player_position(player, (5, 5))

    # Make the player face the target resource
    player_utils.set_player_facing(player, (1, 0))

    # Add the required tool to the player's inventory
    player_utils.set_player_inventory_item(player, "wood_pickaxe", 1)

    # Convert the configured world back to a serializable WorldState
    return export_world_state(world, view=(9, 9))
```

Listing 2: Example of programmatic state manipulation to create an initial state for a scenario. World setup utilities create the environment, while player state utilities configure the agent.

C MUTATORS

Mutators are a core component of our evaluation framework, designed to test a world model's ability to distinguish between plausible and implausible future states, as described in Sec. 4. A mutator is a deterministic function that takes a state-action pair (s_t, a_t) and produces an alternative, incorrect next state $\tilde{s}_{t+1}$. These generated states, called distractors, represent violations of the environment's true dynamics. For example, a distractor might show the agent crafting an item without the necessary resources or moving through a solid obstacle.

By creating a candidate set containing the true next state s_{t+1} and several such distractors $\{\tilde{s}_{t+1}\}$, we construct a discriminative task for the world model. A model with a robust understanding of the environment's laws should assign a significantly higher probability to the true outcome than to any of the distractors. This allows us to quantitatively measure the model's predictive judgment using the state ranking metrics from Sec. 4.

All mutators adhere to a common interface, shown in Listing 3. Each mutator implements a 'precondition' method that checks if the mutation is applicable to a given state and action. If the precondition is met, the 'effect' method is called to generate the mutated state. This design allows for the creation of targeted mutators that only apply under specific circumstances, leading to more subtle and challenging distractors.

```python
class Mutator:
    """A protocol for functions that generate distractor states."""

    def precondition(self, state: WorldState, action: Action) -> bool:
        """
        Returns True if the mutator can be applied to the given
        state-action pair, False otherwise.
        """
        ...

    def __call__(self, state: WorldState, action: Action) -> WorldState:
        """
        Applies a mutation to a copy of the state and returns the
        modified state, representing an illegal transition outcome.
        """
        ...
```

Listing 3: The general interface for a mutator. Each mutator is a callable object with a method to check for applicability.

We have implemented a suite of mutators for the Crafter-OO environment, categorized by the type of game mechanic they target. Tab. 3 provides a comprehensive list of these mutators and the specific rule violations they introduce.

Below we provide detailed descriptions and simplified implementations for three representative mutators from different categories.

Table 3: Catalogue of mutators implemented for the Crafter-OO environment.

Category	Mutator Name	Description of Rule Violation
Physics	`IllegalMovementMutator`	Causes the player to move when a non-movement action is taken.
	`EntityPositionMutator`	Teleports non-player entities to random distant locations.
Combat	`PlayerHealthMutator`	Arbitrarily adds or subtracts a small amount of health from the player.
	`EntityHealthMutator`	Sets the health of non-player entities to a random, incorrect value.
Crafting	`CraftIllegalItemMutator`	Produces a different item than the one specified by the crafting action.
Collection	`CollectIllegalMaterialMutator`	Adds an incorrect resource to the player's inventory when collecting.
Placement	`PlaceIllegalItemMutator`	Places a different object or tile than the one specified by the action.
Player State	`InventoryMutator`	Randomizes all quantities in the player's inventory.

Illegal Movement Mutator

This mutator tests the model's understanding of which actions cause player movement. It activates when the agent takes an action that should not result in a change of position, such as noop or do. The effect is to move the player one step in a random direction, creating a state that would be valid for a movement action but is invalid for the action actually taken. Listing 4 shows its logic.

```
NON_MOVEMENT_ACTIONS = {"noop", "do", "sleep", "make_wood_pickaxe", ...}
DIRECTIONS = [(0, 1), (1, 0), (0, -1), (-1, 0)]

class IllegalMovementMutator:
    def precondition(self, state: WorldState, action: Action) -> bool:
        # This mutator applies only to actions that should not cause movement.
        return action in NON_MOVEMENT_ACTIONS

    def __call__(self, state: WorldState, action: Action) -> WorldState:
        mutated_state = state.model_copy(deep=True)

        # Choose a random direction and update the player's position.
        random_direction = random.choice(DIRECTIONS)
        mutated_state.player.position.x += random_direction[0]
        mutated_state.player.position.y += random_direction[1]

        return mutated_state
```

Listing 4: Simplified logic for the `IllegalMovementMutator`.

Craft Illegal Item Mutator

This mutator targets the logic of crafting recipes. It checks if the agent is attempting to craft an item. If so, it alters the outcome by giving the player a different, randomly selected craftable item. This tests whether the world model has correctly associated specific crafting actions with their unique outcomes. For example, if the action is make_wood_pickaxe, this mutator might instead add a `stone_sword` to the player's inventory. Listing 5 illustrates this process.

```
CRAFTING_ACTIONS = {"make_wood_pickaxe", "make_stone_sword", ...}

class CraftIllegalItemMutator:
    def precondition(self, state: WorldState, action: Action) -> bool:
        # This mutator applies only to crafting actions.
        return action in CRAFTING_ACTIONS

    def __call__(self, state: WorldState, action: Action) -> WorldState:
        mutated_state = state.model_copy(deep=True)

        # Select a different crafting action to determine the illegal outcome.
        other_crafting_actions = CRAFTING_ACTIONS - {action}
        illegal_action = random.choice(list(other_crafting_actions))

```

```python
        # Add the item corresponding to the illegal action to the inventory.
        if illegal_action == "make_stone_sword":
            mutated_state.player.inventory.stone_sword += 1
        # ... logic for other craftable items

        return mutated_state
```

Listing 5: Simplified logic for the `CraftIllegalItemMutator`.

Entity Health Mutator

This mutator introduces arbitrary changes to the health of non-player characters (NPCs), violating the rules of combat, regeneration, and damage. It is an "always on" mutator, meaning its precondition is always true, as health can be a dynamic property in any state. Its effect is to iterate through all non-player entities and set their health to a random value that is not close to their current health. This prevents generating trivial changes that might occur naturally (e.g., from regeneration) and creates a more distinctively incorrect state. Listing 6 shows the implementation.

```python
class EntityHealthMutator:
    def precondition(self, state: WorldState, action: Action) -> bool:
        # This mutator is always applicable.
        return True

    def __call__(self, state: WorldState, action: Action) -> WorldState:
        mutated_state = state.model_copy(deep=True)

        for entity in mutated_state.objects:
            # Skip the player entity.
            if entity.entity_id == mutated_state.player.entity_id:
                continue

            # Generate a new health value that is not the same as the current
            # health, nor immediately adjacent to it.
            possible_health_values = set(range(11)) # Health is 0-10
            excluded_values = {entity.health, entity.health - 1, entity.health + 1}
            valid_new_values = list(possible_health_values - excluded_values)

            if valid_new_values:
                entity.health = random.choice(valid_new_values)

        return mutated_state
```

Listing 6: Simplified logic for the `EntityHealthMutator`.

D Scenarios

An evaluation framework that relies on data from unguided exploration may not sufficiently cover all of an environment's mechanics, especially those that are rare or require specific preconditions. To ensure a comprehensive and targeted assessment of a world model's understanding, we generate evaluation data from a suite of **scenarios**. Each scenario is a short, programmatic interaction sequence designed to isolate and test a single game mechanic under controlled conditions. This approach produces a dataset of transitions that robustly covers the environment's dynamics, from basic resource collection to complex combat encounters. The transitions generated by these scenarios form the basis for the evaluation metrics described in Sec. 4.

D.1 Scenario Structure and Execution

A scenario is defined by a common programmatic interface, as outlined in listing 7. It specifies an initial state, a scripted policy to guide the agent's actions, and a termination condition based on either achieving a specific goal or reaching a maximum number of steps. The execution of a scenario, shown in listing 8, produces a sequence of (state, action, next_state) transitions that serve as ground truth test cases for the world model.

```python
class Scenario:
    @property
    def name(self) -> str: ...

    def get_initial_state(self) ->
     WorldState: ...

    def policy(self, state: WorldState) ->
      Action: ...

    def goal_test(self, transitions: list)
      -> bool: ...

    @property
    def max_steps(self) -> int: ...
```

Listing 7: Structure of an evaluation scenario.

```python
def run_scenario(scenario):
    transitions = []
    state = scenario.get_initial_state()
    for _ in range(scenario.max_steps):
        action = scenario.policy(state)
        next_state = env.transition(state,
     action)
        transitions.append((state, action,
     next_state))
        state = next_state
        if scenario.goal_test(transitions)
    :
            break
    return transitions
```

Listing 8: Execution loop for generating transitions.

D.2 Implemented Scenarios

We developed over 40 scenarios for Crafter-OO, covering every core game mechanic present in the original Crafter environment. These scenarios are categorized by the type of mechanic they test, as detailed in Tab. 4. For many mechanics, we include both a "successful" and an "unsuccessful" variant. The successful version sets up the preconditions for an action to succeed (e.g., having enough resources to craft an item), while the unsuccessful version deliberately violates a precondition. This allows us to test whether a world model understands not only what should happen, but also what should *not* happen.

E Evaluation Implementation Details

This section provides a procedural specification of our evaluation framework. We begin by defining a general-purpose interface that any world model must satisfy to be evaluated. We then detail the computational steps that transform the raw outputs of a model satisfying this interface into the final State Fidelity and State Ranking metrics presented in Sec. 4. The process relies on the evaluation trajectories generated from Scenarios (Appendix D) and the distractor states generated by Mutators (Appendix C).

Our evaluation framework is designed to be model-agnostic. Any world model can be benchmarked, provided it adheres to the simple, two-method interface shown in listing 9. This interface cleanly separates the two core capabilities required for our metrics: the ability to generate a likely future state (for fidelity) and the ability to score a given future state (for ranking).

```python
class EvaluatableWorldModel(Protocol):
    """A protocol for world models that can be evaluated by our framework."""

    def sample_next_state(self, current_state: WorldState, action: Action) -> WorldState
     :
        """
        Generative function: Samples a single predicted next state s_hat_{t+1}
        from the model's posterior distribution P(s_{t+1} | s_t, a_t).
        """
        ...

    def evaluate_log_probability(
        self, state: WorldState, action: Action, next_state: WorldState
    ) -> float:
        """
        Discriminative function: Computes the log-probability of a specific
        next_state given the current state and action.
```

Table 4: Complete list of evaluation scenarios used to test world models in Crafter-OO.

Category	Scenario Name	Description
Movement	random_movement	Tests basic player movement in the cardinal directions.
Collection	collect_wood	Player faces a tree and collects wood.
	collect_drink	Player faces water and collects it.
	collect_stone	Player collects stone with the required pickaxe.
	unsuccessful_collect_stone	Player attempts to collect stone without the required pickaxe.
	collect_coal	Player collects coal with the required pickaxe.
	unsuccessful_collect_coal	Player attempts to collect coal without the required pickaxe.
	collect_iron	Player collects iron with the required pickaxe.
	unsuccessful_collect_iron	Player attempts to collect iron without the required pickaxe.
	collect_diamond	Player collects diamond with the required pickaxe.
	unsuccessful_collect_diamond	Player attempts to collect diamond without the required pickaxe.
	eat_plant	Player eats a ripe plant to gain food.
	unsuccessful_eat_plant	Player attempts to eat an unripe plant.
Crafting	craft_wooden_pickaxe	Player crafts a wooden pickaxe with sufficient wood.
	unsuccessful_craft_wooden_pickaxe	Player attempts to craft without sufficient wood.
	craft_wooden_sword	Player crafts a wooden sword with sufficient wood.
	unsuccessful_craft_wooden_sword	Player attempts to craft without sufficient wood.
	craft_stone_pickaxe	Player crafts a stone pickaxe with required resources.
	unsuccessful_craft_stone_pickaxe	Player attempts to craft without required resources.
	craft_stone_sword	Player crafts a stone sword with required resources.
	unsuccessful_craft_stone_sword	Player attempts to craft without required resources.
	craft_iron_pickaxe	Player crafts an iron pickaxe with required resources.
	unsuccessful_craft_iron_pickaxe	Player attempts to craft without required resources.
	craft_iron_sword	Player crafts an iron sword with required resources.
	unsuccessful_craft_iron_sword	Player attempts to craft without required resources.
Placement	place_table	Player places a crafting table with sufficient wood.
	unsuccessful_place_table	Player attempts to place a table without sufficient wood.
	place_stone	Player places stone with sufficient inventory.
	unsuccessful_place_stone	Player attempts to place stone without sufficient inventory.
	place_furnace	Player places a furnace with sufficient stone.
	unsuccessful_place_furnace	Player attempts to place a furnace without sufficient stone.
	place_plant	Player places a sapling on a grass tile.
	unsuccessful_place_plant	Player attempts to place a sapling without one in inventory.
Combat	zombie_defeat	Player, equipped with a sword, defeats a zombie.
	defeat_skeleton	Player defeats a skeleton.
	eat_cow	Player defeats a cow to obtain food.
	player_death	Player with low health is defeated by a zombie.
NPC Behavior	cow_movement	Tests the stochastic movement of a cow over several steps.
	wake_up	Player goes to sleep and wakes up after their energy is restored.

```
        """
        ...
```

Listing 9: The interface any world model must implement to be compatible with our evaluation framework.

E.1 State Comparison via Canonical Representation

All metrics that involve comparing two world states, such as edit distance or checking for equality, require a deterministic and canonical representation of the state. A direct object-to-object comparison can be unreliable due to factors like in-memory object identifiers or the ordering of elements in lists. To address this, we serialize each `WorldState` object to a canonical JSON format before any comparison is performed. This process, outlined in listing 10, ensures that two states are considered identical if and only if they represent the same game-world configuration.

```
def to_canonical_json(state: WorldState) -> dict:
    """
    Serializes a WorldState object to a deterministic JSON representation.
    """
    # 1. Exclude non-semantic or non-deterministic fields from serialization.
    excluded_fields = {"event_bus", "serialized_random_state"}
    serialized_state = state.model_dump(exclude=excluded_fields, mode="json")
```

```python

    # 2. Sort lists of objects by a stable, unique key to ensure order invariance.
    # The player object is handled separately and removed from the main list.
    serialized_state["objects"] = [
        obj for obj in serialized_state["objects"] if obj["name"] != "player"
    ]
    serialized_state["objects"].sort(key=lambda obj: obj["entity_id"])

    # Chunks are also sorted to ensure map representation is stable.
    if "chunks" in serialized_state:
        serialized_state["chunks"].sort(key=lambda chunk: chunk["chunk_key"])

    return serialized_state
```

Listing 10: Canonical serialization of a WorldState object.

E.2 State Fidelity Metric Calculation

The state fidelity metrics measure the difference between a world model's predicted next state and the ground truth. We use JSON Patch (Bryan & Nottingham, 2013), a standard for describing changes in a JSON document, to provide a precise, interpretable measure of this difference. The calculation for a single transition (s_t, a_t, s_{t+1}) proceeds as described in listing 11.

```python
def calculate_state_fidelity(world_model, s_t, a_t, s_t_plus_1):
    """
    Computes Raw and Normalized Edit Distance for a world model's prediction.
    """
    # 1. Generate a predicted next state from the world model.
    s_hat_t_plus_1 = world_model.sample_next_state(s_t, a_t)

    # 2. Convert both true and predicted next states to canonical JSON.
    json_true = to_canonical_json(s_t_plus_1)
    json_predicted = to_canonical_json(s_hat_t_plus_1)

    # 3. Compute the JSON Patch from the predicted state to the true state.
    patch = jsonpatch.make_patch(json_predicted, json_true)

    # 4. Raw Edit Distance is the number of operations in the patch.
    raw_edit_distance = len(list(patch))

    # 5. Normalized Edit Distance is the raw distance divided by the total number
    # of elements in the true state, providing a scale-invariant measure.
    total_elements = count_elements(json_true)
    normalized_edit_distance = raw_edit_distance / total_elements if total_elements > 0
     else 0

    return raw_edit_distance, normalized_edit_distance
```

Listing 11: Calculation of State Fidelity metrics for a single transition.

Example. Consider a transition where the player, at position $(x = 5, y = 5)$ with $health = 9$, takes the action `move_right`. The true next state, s_{t+1}, has the player at $(x = 6, y = 5)$ with $health = 9$. Suppose a world model predicts a state, $\hat{s}_{t+1}$, where the player correctly moves to $(x = 6, y = 5)$ but their health incorrectly drops to 8.

The simplified canonical JSON representations for the player object in each state would be:

```json
{
  "player": {
    "position": {"x": 6, "y": 5},
    "health": 9
  }
}
```

Listing 12: Canonical JSON for the true next state.

```json
{
  "player": {
    "position": {"x": 6, "y": 5},
    "health": 8
  }
}
```

Listing 13: Canonical JSON for the predicted next state.

The JSON Patch required to transform the predicted JSON into the true JSON is a single `replace` operation: `[{‘‘op’’: ‘‘replace’’, ‘‘path’’: ‘‘/player/health’’, ‘‘value’’: 9}]`. The Raw Edit Distance is the number of operations in this patch, which is 1. The Normalized Edit Distance would be this value divided by the total number of elements in the true state's full JSON representation.

E.3 State Ranking Metric Calculation

State ranking metrics evaluate a model's ability to distinguish the true outcome of an action from a set of plausible but incorrect alternatives. This process involves generating a set of candidate states and using the world model to score them, as detailed in listing 14.

```python
def calculate_state_ranking(world_model, s_t, a_t, s_t_plus_1, mutators, num_distractors
    ):
    """
    Computes Rank@1 and Mean Reciprocal Rank for a world model.
    """
    # 1. Generate a set of distractor states using the mutator bank.
    distractors = []
    applicable_mutators = [m for m in mutators if m.precondition(s_t, a_t)]
    random.shuffle(applicable_mutators) # Ensure variety in distractors
    for mutator in applicable_mutators:
        if len(distractors) >= num_distractors:
            break
        distractors.append(mutator(s_t, a_t))

    # 2. Form the candidate set, including the ground truth and distractors.
    candidate_set = [s_t_plus_1] + distractors
    random.shuffle(candidate_set) # Avoid biasing models that may be sensitive to order

    # 3. Score each candidate state using the world model's log-probability function.
    scores = []
    for s_candidate in candidate_set:
        log_prob = world_model.evaluate_log_probability(s_t, a_t, s_candidate)
        scores.append(log_prob)

    # 4. Determine the rank of the true next state.
    # Ranks are 1-indexed, with rank 1 being the highest score.
    ranked_indices = sorted(range(len(scores)), key=lambda i: scores[i], reverse=True)
    true_state_index = candidate_set.index(s_t_plus_1)
    rank_of_true_state = ranked_indices.index(true_state_index) + 1

    # 5. Calculate metrics from the rank.
    rank_at_1 = 1.0 if rank_of_true_state == 1 else 0.0
    reciprocal_rank = 1.0 / rank_of_true_state

    return rank_at_1, reciprocal_rank
```

Listing 14: Calculation of State Ranking metrics for a single transition.

Example. Continuing the previous example, the true state s_{t+1} is the player moving right. A mutator might generate a distractor state $s_{\text{distractor}}$ where the player illegally teleports to $(x = 20, y =$

20). The candidate set becomes $\{s_{t+1}, s_{\text{distractor}}\}$. A good world model should assign a much higher probability to the true outcome. For instance, it might yield log-probabilities of $\log p(s_{t+1}|\dots) = -0.7$ and $\log p(s_{\text{distractor}}|\dots) = -15.4$. Since $-0.7 > -15.4$, the true state is ranked first. This yields a Rank@1 of 1.0 and a Mean Reciprocal Rank of $1/1 = 1.0$ for this transition.

E.4 Aggregation Across Scenarios

The final metrics reported in Tab. 1 are aggregated from the per-transition results. To ensure that each distinct game mechanic contributes equally to the final score, we employ a two-level aggregation strategy. First, we compute the mean metric values across all transitions within a single scenario. Second, we compute the final reported metric by taking the mean of these per-scenario means. This prevents scenarios with more transitions (e.g., a long movement sequence) from dominating the overall results compared to scenarios with fewer, more critical transitions (e.g., a single crafting action). listing 15 formalizes this entire pipeline.

```python
def evaluate_world_model(world_model, scenarios, mutators, config):
    """
    Runs the full evaluation pipeline and returns aggregated metrics.
    """
    per_scenario_metrics = {}

    # 1. Evaluate each scenario independently.
    for scenario in scenarios:
        transitions = run_scenario(scenario) # See Sec. C.1 for run_scenario

        scenario_results = []
        for (s_t, a_t, s_t_plus_1) in transitions:
            # Calculate metrics for each transition in the scenario.
            r_at_1, mrr = calculate_state_ranking(
                world_model, s_t, a_t, s_t_plus_1, mutators, config.num_distractors
            )
            raw_ed, norm_ed = calculate_state_fidelity(
                world_model, s_t, a_t, s_t_plus_1
            )
            scenario_results.append({
                "R@1": r_at_1, "MRR": mrr,
                "RawEditDist": raw_ed, "NormEditDist": norm_ed
            })

        # 2. First level of aggregation: average metrics within the scenario.
        if not scenario_results: continue
        per_scenario_metrics[scenario.name] = {
            key: sum(res[key] for res in scenario_results) / len(scenario_results)
            for key in scenario_results[0]
        }

    # 3. Second level of aggregation: average the per-scenario means.
    final_metrics = {
        key: sum(metrics[key] for metrics in per_scenario_metrics.values()) / len(
    per_scenario_metrics)
        for key in list(per_scenario_metrics.values())[0]
    }

    return final_metrics
```

Listing 15: Overall evaluation pipeline and metric aggregation.

F Synthesis and Exploration Implementation Details

The process of generating candidate world laws is divided into two main stages: unguided exploration to collect a dataset of interactions, and law synthesis to propose programmatic laws from that dataset.

F.1 Exploration Policy

To gather the interaction dataset $\mathcal{D} = \{(s_t, a_t, s_{t+1})\}_{t=1}^{N}$, we employ an autonomous exploration policy driven by a large language model. This policy operates without access to environment-specific rewards or human-provided goals. Instead, it is given a high-level instruction to explore the environment and discover as many of its underlying mechanics as possible, treating the task as a reverse-engineering problem. The full prompt provided to the exploration policy is detailed in box F.1.

Box F.1| Exploration Policy Prompt

```
You are an explorer in an unknown digital world. Your mission is to experience as many of the world's hidden mechanics as possible. Your recorded experiences will be analyzed later to create a complete map of the world's physical laws.

The laws of any world can be thought of as IF-THEN hypotheses: `IF (a specific situation occurs) AND (you take an ACTION), THEN (a certain outcome happens).`

To succeed, you must trigger as many different `IF-THEN` scenarios as you can.

**What to Expect in the World:**
This world is complex and may be dangerous.
- **Hostile Entities:** You may encounter creatures that are hostile and will attack you.
- **Resource Collection:** The world contains raw materials that can be gathered, though there may be preconditions for collection.
- **Item Production:** You have the ability to craft useful items from raw materials, though there may be preconditions for production.
- **Combat:** You can engage in combat with the entities you encounter.

Your primary goal is to discover the rules governing these activities.
You will need to explore the game world by moving around and interacting with the entities and materials in the world.
If an action has no effect, you may not have fulfilled the preconditions for the action to have an effect.
Try out a variety of actions from each category: movement, interaction, placement, production.
If an action seems to have no effect, you may not have fulfilled the preconditions for the action to have an effect.
Try to acquire additional resources or change something about the world and try again.
Before taking actions, set goals for yourself in an IF-THEN format, and let the results invalidate those actions.
If an entity is hostile, you can attempt to defend yourself from it.
If an entity seems passive or beneficial, you can attempt to interact with it.
You will likely need to progress through the "tech tree" of the game in a specific order.
This will require interleaving resource collection with placement of crafting stations and production of better tools.
In the meantime, you will need to survive hostile enemies and find ways to heal from damage you've taken.
Some resources likely cannot be acquired without first producing a tool to acquire them.
Tools may require a mix of materials and crafting stations to produce.

The following are the only valid actions you can take:

{action_strings}.

You will now receive observations from the world. Begin your exploration.
```

This LLM-based policy is crucial for gathering sufficiently diverse data in a hostile environment like Crafter-OO. A purely random policy survives for an average of 100 steps before the agent perishes. In contrast, our LLM-based policy navigates the environment for an average of 400 steps. Despite this improvement, exploration remains a significant bottleneck. The policy often struggles to progress through the environment's technology tree, frequently failing to discover the necessary preconditions for crafting advanced items. It also exhibits a tendency to forget previously learned information, which prevents it from effectively building upon past successes within a single trajectory.

F.2 Law Synthesis from Trajectories

The law synthesis pipeline processes the trajectory data from the exploration phase to generate a set of candidate laws $\{L_i\}$. The core idea is to identify state transitions where meaningful changes occur, and then prompt a large language model to propose atomic, programmatic laws that explain those specific changes. This process is outlined in Algorithm 17.

Change Detection for Tractable Synthesis. In an environment with a complex, structured state like Crafter-OO, changes between timesteps are often sparse and localized to specific subcomponents. To make law synthesis tractable, we first isolate these localized changes to provide a focused context for the synthesizer. This is achieved through a set of detectors that monitor different **aspects** of the world state. An aspect is a semantically-cohesive subset of the state, typically corresponding to a top-level attribute (e.g., 'player.inventory') or a collection of entities of the same type (e.g., all 'ZombieState' objects). For each transition (s_t, a_t, s_{t+1}), we check for changes across all aspects. If a detector identifies a change, a synthesis task is created for that specific transition and aspect.

```
class ChangeDetector:
    def aspect_name(self) -> str: ...
    def has_changes(self, s_t: WorldState, s_t_plus_1: WorldState) -> bool: ...

class PlayerInventoryChangeDetector(ChangeDetector):
    def aspect_name(self): return "player_inventory"
    def has_changes(self, s_t, s_t_plus_1):
        return s_t.player.inventory != s_t_plus_1.player.inventory

class ZombieStateChangeDetector(ChangeDetector):
    def aspect_name(self): return "zombies"
    def has_changes(self, s_t, s_t_plus_1):
        # Logic to compare zombie states between s_t and s_t_plus_1
        ...

# A list of all detectors is used to check each transition
ALL_DETECTORS = [
    PlayerInventoryChangeDetector(),
    ZombieStateChangeDetector(),
    ... # Other detectors for map tiles, cows, etc.
]
```

Listing 16: Simplified change detection logic. Each detector checks for changes in a specific part of the world state between s_t and s_{t+1}.

This decomposition is not a form of environment-specific guidance but rather a generic mechanism derived directly from the structure of the state representation itself. The Crafter-OO environment exposes an object-oriented state, defined by a schema of classes and attributes. Our change detectors mirror this schema, creating one detector for each top-level attribute and for each object type. This approach provides a structural inductive bias—that the environment's causal mechanisms are likely aligned with its object-oriented structure—without embedding knowledge of the environment's actual dynamics. The process could be fully automated for any environment that exposes a typed, structured state; the detectors can be generated programmatically by reflecting on the state schema. This is analogous to how a computer vision model might process distinct objects in a scene separately; we partition the state space based on its given structure, but the rules governing the interactions between these partitions must still be learned from scratch.

Prompt Generation. For each transition-aspect pair that triggers a synthesis task, we generate a detailed prompt for the LLM. The goal is to provide all necessary context for the model to infer the underlying game mechanic. The prompt contains several key components:

1. The initial state s_t and resulting state s_{t+1}, serialized to a structured format (JSON).
2. The action a_t that caused the transition.
3. A textual ‘diff‘ that highlights the exact changes between s_t and s_{t+1}.
4. A human-readable 2D ASCII rendering of the local environment around the player for both states, providing spatial context.
5. The name of the aspect (e.g., “player_inventory”) that changed, which instructs the LLM to focus its analysis.

This structured presentation of the transition allows the LLM to ground its reasoning in the specific, observed changes. The full prompt template is provided in box F.2.

Box F.2| Synthesis Prompt

```
## Role
You are a **World Law Synthesizer** - an expert at analyzing game state transitions and extracting the underlying rules that govern virtual worlds. Your job is to observe how actions transform game states and codify these transformations into precise, executable laws that can model game mechanics, as well as try to model aspects of the underlying transition dynamics as functions.

## Task Description
Given a world state, an action taken, an aspect of the state we are interested in modeling, and the resulting next world state (plus a diff highlighting the changes), you must:
- Identify how the aspect of the state we are interested in modeling changed between the observations
- Determine the underlying rules or laws that caused these changes
- Implement these laws as executable Python code using the provided WorldState interface and DiscreteDistribution for predictions

**IMPORTANT: You should write MULTIPLE laws when you observe multiple distinct changes.** Each law you write should be modular, minimalistic, focused on a single game mechanic, and capable of being combined with other laws to model complex game behavior.

In particular, you should strive to write laws that are responsible for as little of the state as possible. In any given transition, you may see many changes. Each of these changes could be caused by a different law. Think about what changes could be grouped together into a single law, and write separate laws for different types of changes.

- Break up the laws to each account for a single precondition and effect. For example, if an entity moves, write a law for the movement of entities of that type. If a player takes a particular action, write a law for that action specifically.
- Certain attributes cannot have a `DiscreteDistribution` applied to them. For example, the `materials` field should just be modified directly, not wrapped in a `DiscreteDistribution`. Alternatively, use `set_material` or `set_facing_material` to modify the materials field. Either way, they cannot be wrapped in a `DiscreteDistribution`.
- Use the `DiscreteDistribution` class to indicate probabilistic predictions, for example when trying to write a general law governing all entities of a type when you cannot reconcile all changes visible to that entity type into a deterministic law.
- You DO NOT need to use imports. Everything you need can be coded without the use of imports, and all classes defined below are already imported.

```

````
## Aspect of the State
You will be given an aspect of the state we are interested in modeling. The laws you write should be focused on modeling changes to this aspect of the state.
However, you can use _all_ of the state to help you write the laws, as the aspect of the state may be influenced by other aspects of the state.
For example, if told to focus on Zombies, you should write laws that govern the behavior of Zombies. This behavior may be influenced by other parts of the state such as the player's actions or position.
If told to focus on the player, you should write laws that model how the player's state changes. Again, these effects may be influenced by the entities that the player is interacting with.

## Guidelines for Writing Laws
- Some laws may be dependent on an action being taken, or a particular state of the world, while others may always apply. For these, the precondition can always be `True`.
- Make use of `adjacent_to_player` and `get_target_tile` to help you write laws about interactions between the player and other entities.
- Do NOT use `entity_id` when writing laws. You should instead write laws that apply to a type of entity, e.g. `ZombieState` or `CowState`.
- When modifying attributes, use RELATIVE assignments rather than absolute assignments. For example, instead of changing a entity's position via `entity.position.x = DiscreteDistribution(support=[7])`, use `entity.position.x = DiscreteDistribution(support=[entity.position.x + delta])`. The only exception to this is when modifying the materials field.
- Use the helper functions `get_object_of_type_in_update_range`, and `get_objects_in_update_range` rather than writing your own iteration logic.
- You DO NOT need to use the `entity_id` attribute. Use `get_target_tile` to get the tile or entity targeted by the player. Use `adjacent_to_player` to check if an entity is adjacent to the player for interactions between the player and other entities.
- Consider writing laws that make ”soft” predictions. For example, if you see an entity moving but are unsure if it is a general principle, you can assign a discrete distribution to the entity's position to represent your uncertainty. Example: `entity.position.x = DiscreteDistribution(support=[entity.position.x + delta_a, entity.position.x - delta_b, ...])`.
- You can speculatively pose laws, but these should go last. Speculative laws are those that were not directly observed in the transition, but those that you believe might exist. For example, given that you have identified a law about a certain crafting recipe, you can speculatively pose a law about _other_ crafting recipes that you believe might exist.



## Formatting Instructions
Structure your response exactly as follows. **You can write multiple laws by repeating the pattern below for each law:**

```xml
<keyChanges>
List the specific, concrete changes that occurred between the observations:
- What entities appeared, disappeared, or moved
- What stats/values changed and by how much
- What items were added/removed from inventory
- Any other measurable state differences
</keyChanges>
<naturalLanguageLaw>
Write a clear, concise description of the game rule that explains these changes:
- What triggers this law (the preconditions)
- What the law does (the effects/transformations)
- Any important parameters or variations
- Give the law a descriptive name
</naturalLanguageLaw>
````
````

````
<lawCode>
```python
class YourLawNameHere:
    def __init__(self, param1: type = default_value, param2: type = default_value):
        """Initialize with configurable parameters."""
        self.param1 = param1
        self.param2 = param2
        # Add any lookup tables or constants here

    def precondition(self, current_state: WorldState, action: str) -> bool:
        """Return True if this law should apply to the given state and action."""
        # Implement your precondition logic here
        # Check action type, entity presence, player state, etc.
        return False  # Replace with actual logic

    def effect(self, current_state: WorldState, action: str) -> None:
        """Apply the law by modifying the world state."""
        # Implement the state transformation here
        # Modify entities, player stats, inventory, etc.
        # Use DiscreteDistribution(support=[value]) to set deterministic predictions
        # Example: current_state.player.health = DiscreteDistribution(support=[new_health])
        pass  # Replace with actual implementation
```
</lawCode>

<keyChanges>
[Changes for second law...]
</keyChanges>
<naturalLanguageLaw>
[Description of second law...]
</naturalLanguageLaw>
<lawCode>
```python
class YourSecondLawNameHere:
    # [Implementation of second law...]
```
</lawCode>
```

**Critical Formatting Notes**:
- **Write multiple laws when you observe multiple distinct changes** - each law should focus on a single type of change
- Use exactly these XML-style tags: `<keyChanges>`, `<naturalLanguageLaw>`, `<lawCode>`
- Close each tag properly: `</keyChanges>`, `</naturalLanguageLaw>`, `</lawCode>`
- Put all Python code inside triple backticks within the `<lawCode>` section
- Be precise and specific in the key changes - use exact numbers and entity names from the observations
- Make the natural language law description clear enough that another programmer could implement it independently
- Only output the code for the law, not the entire file. Assume the `WorldState` class as well as its components are already defined.
- Format your response well, with newlines between the tags and code blocks.
- **Each law should be completely self-contained** - repeat the full XML structure for each law you write.

## WorldState
The world state is a Pydantic model that represents the complete game world state. The world laws you write will operate on this state.

````
````

````
```python
{{ world_state_schema }}
```

# World Laws
Each world law must conform to the following interface:

```python
class WorldLaw:
    def precondition(self, current_state: WorldState, action: str) -> bool:
        """Return True if this law should apply to the given state and action."""
        ...

    def effect(self, current_state: WorldState, action: str) -> None:
        """Apply the law by modifying the world state."""
        # Use DiscreteDistribution(support=[value]) to set deterministic predictions
        # Example: current_state.player.health = DiscreteDistribution(support=[new_health])
        ...
```
You may add any additional fields or methods to the class as needed.

## DiscreteDistribution Usage
When modifying state values in your law's `effect` method, you must wrap the new values with `DiscreteDistribution`:

```python
# For deterministic predictions:
current_state.some.value = DiscreteDistribution(support=[new_health])

# For stochastic predictions (if needed):
current_state.some_value = DiscreteDistribution(support=[value1, value2, value3])
```

The `DiscreteDistribution` class represents probabilistic predictions over discrete values. For deterministic laws, you typically provide a single value in the support list. For stochastic laws, you provide multiple values in the support list to represent the possible outcomes.

When accessing the materials field, pay attention to the `MaterialT` type. Everything in the `materials` field is a `MaterialT`. Do not use the emojis in the world map, they are only there for your convenience.

# Your Turn
## Aspect of the State
Focus on modeling changes to the following aspect of the state:
{{ aspect_of_state }}

## Focused Changes for {{ aspect_of_state }}
{{ aspect_changes }}

## View Legend
{{ view_legend }}

## State
```json
{{ state }}
```
### Local View
```
{{ local_view }}
```
````

```

## Action
The action taken was: "{{ action }}"
```

Law Generation and Parsing. The generated prompt is sent to an LLM, which is instructed to return one or more atomic laws that explain the observed changes for the specified aspect. An atomic law is a simple, modular rule focused on a single game mechanic. The LLM's response is formatted using XML-style tags to clearly delineate the key components of each proposed law.

The expected format for a single law is:

````
<keyChanges>...</keyChanges>
<naturalLanguageLaw>...</naturalLanguageLaw>
<lawCode>
```python
class LawName:
    def precondition(self, state, action): ...
    def effect(self, state, action): ...
```
</lawCode>
````

We parse this semi-structured text to extract the natural language description and the executable Python code for each proposed law. This is done by searching for the corresponding tags and extracting their content. The Python code is then loaded as a candidate law for the subsequent parameter inference stage.

```python
def synthesize_laws_from_trajectory(trajectory: list[Transition]) -> list[Law]:
    candidate_laws = []

    # Iterate over all transitions from the exploration data
    for transition in trajectory:
        s_t, action, s_t_plus_1 = transition

        # 1. Detect which aspects of the state have changed
        changed_aspects = []
        for detector in ALL_DETECTORS:
            if detector.has_changes(s_t, s_t_plus_1):
                changed_aspects.append(detector.aspect_name())

        # 2. For each detected change, generate laws
        for aspect in changed_aspects:
            # 2a. Render a detailed prompt for the LLM
            prompt = render_synthesis_prompt(
                state=s_t,
                action=action,
                next_state=s_t_plus_1,
                aspect_of_state=aspect
            )

            # 2b. Query the LLM to synthesize laws
            llm_response_text = call_llm(prompt)

            # 2c. Parse the response to extract structured laws
            parsed_laws = parse_laws_from_response(llm_response_text)
            candidate_laws.extend(parsed_laws)

    return candidate_laws
```

Listing 17: High-level overview of the law synthesis pipeline.