ABSTRACT BELIEFS FOR CODE WORLD MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Code world models represent environment dynamics as executable programs. Under partial observability, existing methods maintain beliefs with concrete particles, each committing every hidden variable to a single value. As joint assignments grow combinatorially with the number of hidden variables, covering the plausible states exhausts a fixed particle budget. We introduce the *distributional particle* (d-particle), whose state variables can each hold a distribution over possible values, so *one particle can represent many possible worlds*. Building on d-particles, we propose **Abstract Belief Coder (ABC)**: a shared Factor Library defines distribution families whose operations propagate and condition these distributions and return the likelihoods that weight particles, and an LLM proposes state representations and programs ranked by this filtering likelihood. On six partially observed MiniGrid tasks, ABC succeeds in 90.6% with ground-truth models versus 55.0% for a concrete-particle baseline, and in 75.6% versus 46.1% with LLM-generated models; it also leads on two Crafter tasks.

1 INTRODUCTION

Code world models represent environment dynamics as executable programs that an agent can run forward to predict the outcome of each action. Written by language models that draw on prior knowledge, these programs can be learned from little environment interaction (Tang et al., 2024; Dainese et al., 2024). In many environments, however, the agent cannot observe the full state: limited sensing or occlusion hides part of it. In such a partially observable Markov decision process (POMDP) (Kaelbling et al., 1998), the same action can lead to different outcomes depending on the hidden state, so the agent must predict and plan with a belief over the states consistent with its past observations. How a code world model represents and updates this belief therefore determines which outcomes the agent can anticipate.

Existing code world models fall into two groups. The first assumes that the agent fully observes its current state, so each program maps one known state to the next (Tang et al., 2024; Dainese et al., 2024). The second, including POMDP Coder (Curtis et al., 2025) and Pinductor (Six et al., 2026), maintains the belief by particle filtering, reweighting particles by how well they explain each observation. Each particle is a complete specification of the world with no uncertainty of its own, and uncertainty is expressed only across the set of particles; we call this a *concrete-particle belief*. Even a variable that no observation has revealed must therefore take a single value in every particle: if an unseen key may lie in any of n cells, each particle places it in exactly one, so covering its possible positions takes n particles, and two such keys take n^2.

Beyond the particle budget, this representation fails when several hidden variables must be inferred together. Since each concrete particle fixes every hidden variable, a view that contradicts one object's position drives the whole particle's weight toward zero, discarding its correct guesses about other objects. Each replacement particle must again match all past observations on every hidden variable, which becomes unlikely as hidden variables accumulate. Even with a human-authored ground-truth model, a concrete-particle filter succeeds in every episode with one hidden key but in only 40.0% with three moving balls (Section 4.1), and its belief keeps almost no mass on the true state with three moving balls or twelve doors with hidden locks despite thousands of replacements (Section 4.2).

We therefore hypothesize that a belief can track many hidden variables with few particles if a state variable may hold a distribution over its possible values instead of a single value, so uncertainty lives within particles as well as across them. In Figure 1b, the view hides the agent's position and two keys,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

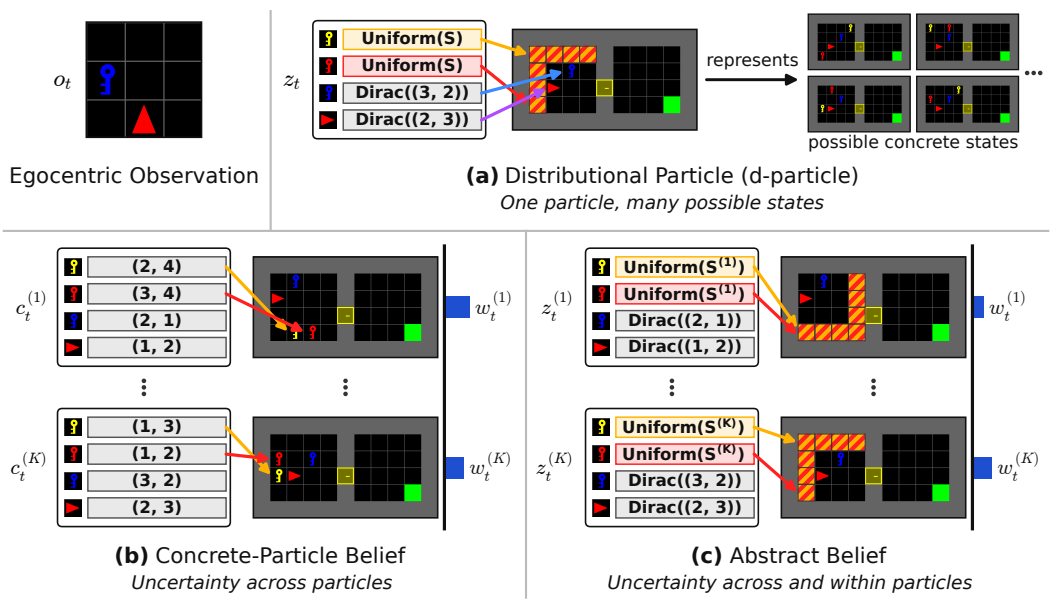


Figure 1: Concrete-particle vs. abstract beliefs. The egocentric observation o_t does not reveal the agent pose and shows neither the yellow nor the red key. (a) A distributional particle (d-particle) with abstract state z_t fixes the agent pose and blue key and holds `Uniform` factors over the striped cells S for the unseen keys, so one particle represents many concrete states. (b) A concrete-particle belief needs one particle per combination of agent pose and key positions, a count that multiplies with every hidden object. (c) An abstract belief needs only one d-particle per pose, with pose hypotheses across particles and key uncertainty within each.

so each concrete particle must pick one combination. A particle that fixes only the pose knows which cells each view rules out, so a view without a key shrinks that key's uniform distribution instead of discarding the particle, and one such particle per pose covers every combination (Figure 1c).

We propose the **Abstract Belief Coder (ABC)**, a code world model whose belief is a weighted set of *distributional particles* (d-particles), each representing every state variable by a distribution. When these distributions stay independent and are updated exactly, the belief is a Rao–Blackwellized particle filter (Doucet et al., 2000; Montemerlo et al., 2002): it samples only the variables that make the rest tractable and integrates those out in closed form, whereas concrete particles sample every variable. Unlike standard Rao–Blackwellization, the split is not given: an LLM proposes each variable's distribution family with the programs that update them, and ABC selects proposals by how well they predict observed trajectories. To keep these likelihoods comparable, the LLM-written programs never update distributions themselves but call the operations of a shared **Factor Library**.

On six partially observed MiniGrid (Chevalier-Boisvert et al., 2023) tasks, ABC improves success over concrete-particle beliefs with both LLM-generated models (75.6% versus 46.1%) and human-authored ground-truth ones (90.6% versus 55.0%). The gain concentrates where concrete particles failed: with ground-truth models and three moving balls, ABC succeeds in 93.3% instead of 40.0%, and its belief keeps probability on the true state in every task. More particles do not close the gap: a single d-particle matches or exceeds thirty concrete particles in every task, and ABC also leads on two Crafter (Hafner, 2021) tasks.

Our contributions are threefold:

1. We identify that existing code world models express uncertainty only across particles, each committing every state variable to a single value, and show that such beliefs lose the true state as hidden variables accumulate.
2. We propose ABC, whose d-particles bring Rao–Blackwellized filtering, precluded by concrete particles, to code world models: an LLM proposes each variable's distribution family, and the operations of a shared Factor Library keep the resulting likelihoods comparable.
3. We show that ABC's gain in MiniGrid comes from the belief representation rather than from model generation or particle count, and that ABC extends to Crafter.

2 World Modeling with Abstract Beliefs

This section formalizes the abstract belief and the components that operate on it, proceeding from definitions (Sections 2.1–2.2) to the factor library (Section 2.3), its use in filtering and planning (Sections 2.4–2.5), and model induction (Section 2.6).

2.1 Separating Representations from Programs

Existing code world models for POMDPs fix the state representation in advance, supplied to the LLM as a code interface, and generate only the programs that operate on it (Curtis et al., 2025; Six et al., 2026). ABC separates the representation from the programs and lets it vary, because which variables should hold distributions depends on the task: a stationary key can keep a uniform distribution over candidate cells, whereas a moving ball needs one that the transition program propagates. Since the representation determines both what the belief can express and what the programs must compute, ABC chooses the two together as a *world-model hypothesis* $\Theta = (\rho, M_\rho)$. The representation ρ specifies the state variables, their types, and the distribution families each may use; the executable model $M_\rho = (I, T, O, R)$ consists of programs that generate initial states (I), predict state changes under actions (T), condition the predicted state on an observation and return its likelihood (O), and compute rewards and termination (R).

2.2 Distributional Particles and Abstract Beliefs

ABC represents its belief as an *Abstract Belief State (ABS)*, a weighted mixture of *distributional particles* (d-particles). Unlike a concrete particle, which fixes a single value for every state variable, a d-particle can hold several possible values of a variable, with their probabilities, in one particle. We first review the concrete-particle belief of prior work and then define d-particles and the ABS.

Concrete-Particle Belief. In prior work (Curtis et al., 2025; Six et al., 2026), the representation ρ is fixed in advance. It specifies the *state variables* $x_1, \ldots, x_n$, such as the agent's position and direction, and the domain $\mathcal{X}_i$ of each. Each particle is a *concrete state* $c = (v_1, \ldots, v_n)$ that assigns a specific value $v_i \in \mathcal{X}_i$ to every variable, and we denote the concrete state space by $\mathcal{C}_\rho = \mathcal{X}_1 \times \cdots \times \mathcal{X}_n$. For example, each `MinigridState` instance (Figure 2a) must specify the full grid, including unobserved cells, and the agent's position, direction, and carried object.

These methods approximate the belief with K particles, each with state $c_t^{(k)} \in \mathcal{C}_\rho$ and weight $w_t^{(k)}$. This *concrete-particle belief* is

$$\hat{b}_t(c) = \sum_{k=1}^{K} w_t^{(k)} \mathbf{1}\left[c = c_t^{(k)}\right], \qquad (1)$$

where the weights are nonnegative and sum to one. Each particle carries no uncertainty of its own, so K particles represent at most K distinct concrete states.

(a) Prior work: a concrete state class

```
@dataclass
class MinigridState:
    grid: NDArray[np.int8]
    agent_pos: Tuple[int, int]
    agent_dir: int
    carrying: Optional[int]
```

(b) ABC: a representation with factors

```
@dataclass
class Agent:
    pos: Dirac[Cell]
    dir: Dirac[Dir]
    carrying: Dirac[Obj]
@dataclass
class Key:
    pos: Dirac[Cell]
       | Uniform[Cell]
```

Figure 2: State representations for MiniGrid. In (b), the key position may hold a `Dirac` or a `Uniform` factor.

Abstract Belief. A d-particle instead represents each state variable x_i by a probability distribution μ_i over $\mathcal{X}_i$, stored as a *factor* $F = (f, \mathcal{U}, q)$: its *factor family* f fixes the form of the distribution, its domain $\mathcal{U} \subseteq \mathcal{X}_i$ fixes the possible values, and its parameters q select one distribution of that form. For example, a point mass at a specific value v is the factor $(\texttt{Dirac}, \mathcal{U}, v)$, and a key whose cell is unknown can hold the factor $(\texttt{Uniform}, \mathcal{U}, S)$, which assigns probability $1/|S|$ to each cell in the set $S \subseteq \mathcal{U}$ of cells that may still contain the key. The d-particle's *abstract state* $z = (\mu_1, \ldots, \mu_n)$ defines a distribution over $\mathcal{C}_\rho$ as $\mu_\rho(c \mid z) = \prod_{i=1}^{n} \mu_i(v_i)$ for $c = (v_1, \ldots, v_n)$; dependencies between variables are captured across d-particles.

The representation ρ also specifies which families each variable may use (Figure 2b). Under this representation, a d-particle may place the agent at $(2, 3)$ with a `Dirac` factor and keep the key's location as the `Uniform` factor above. Once an observation reveals the key, conditioning shrinks S to that single cell, a point mass, without changing ρ.

ABC approximates the belief with K d-particles, each with abstract state $z_t^{(k)}$ and weight $w_t^{(k)}$. Their weighted mixture is the ABS:

$$\hat{b}_t(c) = \sum_{k=1}^{K} w_t^{(k)} \mu_\rho\Big(c \mid z_t^{(k)}\Big), \tag{2}$$

where the weights are nonnegative and sum to one. If every μ_i in a d-particle is a point mass at v_i, then $\mu_\rho(c \mid z) = \mathbf{1}[c = (v_1, \ldots, v_n)]$; when this holds for all d-particles, Equation 2 reduces to the concrete-particle belief in Equation 1.

2.3 Factor Library

ABC generates multiple world-model hypotheses and must determine which best explains the interactions. If each hypothesis implemented its own probabilistic updates, differences could reflect how hypotheses compute probabilities rather than how well they model the environment. ABC therefore separates the two: the programs T and O of each hypothesis model the environment, while all hypotheses share one library for probability computation, the **Factor Library**. It defines the factor families that ρ may declare (Section 2.2) and the operations that update them.

The library currently contains five families, each with its own parameters: `Dirac` (a single value), `Uniform` (a set of possible values), `Categorical` (a probability vector), `Bernoulli` (the probability of being true), and `Gaussian` (a mean and a covariance). For each family, the library provides two operations on a factor F. *Propagation* predicts the factor after an action, using a transition kernel τ from T that specifies how the variable can change and with what probabilities. *Conditioning* then reweights and normalizes the prediction, using a likelihood function η from O that specifies how likely the observation is under each possible value $u \in \mathcal{U}$:

$$F_{\text{pred}} = \text{Propagate}(F, \tau), \qquad (F_{\text{post}}, \ell) = \text{Condition}(F_{\text{pred}}, \eta), \qquad \ell = \mathbb{E}_{u \sim F_{\text{pred}}}[\eta(u)]\,. \tag{3}$$

Here, F_{pred} and F_{post} are the predicted and posterior factors, and ℓ is the marginal likelihood of the evidence for this factor, from which Section 2.4 computes the d-particle weights.

Both operations change only a factor's parameters, never its family or domain. Each family accepts only kernels under which it is closed and likelihoods conjugate to it (e.g., linear-Gaussian models for `Gaussian`, indicator likelihoods for `Uniform`), so both operations are exact and closed-form; all others are rejected rather than approximated. Any approximation must be computed explicitly in O, outside the library. Appendix C gives the update rules of each family, and Appendix B.1 the interface with M_ρ.

2.4 Filtering Abstract Beliefs

ABC filters by updating the distribution that each d-particle holds, rather than by sampling its factors. Starting from d-particles generated by I, T predicts each abstract state after an action, and O conditions the prediction on the new observation:

$$\bar{z}_t^{(k)} = T\big(z_{t-1}^{(k)}, a_{t-1}\big), \qquad \big(z_t^{(k)}, \ell_t^{(k)}\big) = O\big(\bar{z}_t^{(k)}, o_t\big). \tag{4}$$

T and O update factors through Propagate and Condition (Section 2.3). O returns the posterior abstract state $z_t^{(k)}$ and the likelihood $\ell_t^{(k)}$ of o_t, which combines the evidence that Condition returns per factor (Appendix B.2).

Each weight is then multiplied by this likelihood and normalized:

$$L_t = \sum_{k=1}^{K} w_{t-1}^{(k)} \ell_t^{(k)}, \qquad w_t^{(k)} = \frac{w_{t-1}^{(k)} \ell_t^{(k)}}{L_t}. \tag{5}$$

Here, L_t is the predictive likelihood of o_t under the current world model and belief, which Section 2.6 uses to score hypotheses. When the factors stay independent and every factor update is exact,

$\ell_t^{(k)}$ marginalizes over the factors instead of evaluating one sample, so ABC is a Rao–Blackwellized particle filter: variables with non-`Dirac` factors are marginalized analytically, and only the rest are sampled. D-particles with zero likelihood are discarded, and identical posterior abstract states are merged by summing their weights; Appendix B.2 also describes reinitialization when all likelihoods are zero.

2.5 Planning with Abstract Beliefs

ABC plans open-loop over the ABS: it searches over action sequences by applying T and R to every d-particle, without simulating observations. Expanding an action a from a search node gives, for each of its d-particles $z^{(k)}$, a successor $\tilde{z}^{(k)}$ together with a reward $r^{(k)}$ and a termination flag $d^{(k)}$:

$$\tilde{z}^{(k)} = T\big(z^{(k)}, a\big), \qquad \big(r^{(k)}, d^{(k)}\big) = R\big(z^{(k)}, a, \tilde{z}^{(k)}\big). \tag{6}$$

The successors keep their parents' weights $w^{(k)}$ and form the next node. A best-first search accumulates the cost $c_a - \bar{r}$, where $\bar{r} = \sum_k w^{(k)} r^{(k)}$ is the weighted predicted reward and c_a a constant action cost. ABC executes only the first action of the selected sequence, updates the ABS with the resulting observation (Section 2.4), and replans.

An interaction that depends on a factor, such as picking up an unseen key, would split every successor over the factor's support. T instead evaluates it at one representative value (Appendix B.3) and never stores it, so the factor remains a distribution. For example, if a doorway is equally likely to occupy three adjacent cells, T uses the middle cell. Filtering uses the same placement, which approximates the transition but not the factor updates; the next observation then corrects the outcome.

2.6 Inducing World-Model Hypotheses

So far, we have described how ABC filters and plans with ρ and M_ρ. In ABC, an LLM writes both as world-model hypotheses from a task description, offline trajectories, and the Factor Library documentation, without access to hidden states or ground-truth models (Appendix D). These hypotheses may use different representations ρ, but they are all evaluated on the same observations. To choose among them, ABC therefore runs the filter of Section 2.4 on each offline trajectory h and scores each hypothesis by the likelihood of the observations:

$$s_h = \sum_t \log L_t - \lambda\, m_h, \tag{7}$$

averaged over trajectories, similar to Pinductor's score (Six et al., 2026) for a fixed representation. The first term sums the log predictive likelihoods L_t of Equation 5. The second, in the spirit of minimum description length (Rissanen, 1978), charges λ for each of the m_h predicates the belief stores across its d-particles, such as the agent or a room, favoring shorter descriptions when likelihoods tie.

Yet the score only ranks hypotheses: one whose T never propagates a moving ball's factor cannot explain a later view of the ball outside the factor's support and scores low, but nothing in the score points to T. To locate such errors, ABC takes a top-scoring hypothesis and checks each offline trajectory for the first observation it cannot explain. If such an observation exists, the LLM receives the trajectory up to it and revises the hypothesis, which is scored and joins the candidates for the next check. Once a checked hypothesis explains every offline trajectory, it is frozen and evaluated without further LLM calls.

3 Related Work

LLMs have built executable world models from interaction data (Tang et al., 2024; Dainese et al., 2024; Piriyakulkij et al., 2026). Under partial observability, POMDP-Coder and Pinductor (Curtis et al., 2025; Six et al., 2026) maintain particles that each specify a complete state, so alternative hidden-state assignments require separate particles. For a supplied model, Rao–Blackwellized filtering (Doucet et al., 2000; Montemerlo et al., 2002) and automatic marginalization (Murray et al., 2018; Atkinson et al., 2022) retain analytic distributions within particles, while object-oriented POMDPs factor beliefs over objects under independence assumptions (Wandzel et al., 2019; Mangannavar et al., 2024). ABC makes the belief representation part of model induction: an LLM pro-

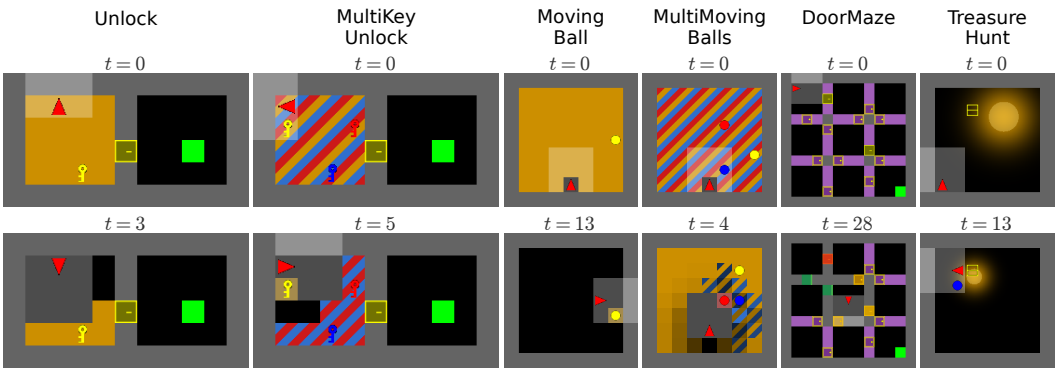


Figure 3: MiniGrid tasks and the d-particles of ABC. Each panel draws one d-particle of ABC's belief at step t of one episode per column with the ground-truth model, so all its factors belong to that particle; Unlock and MultiKeyUnlock at $t = 0$ instead average the factors of ten d-particles that differ in the agent's position. Each object's factor is shaded in its color, brighter where more probable and striped where factors overlap; a fainter shade marks a located object, and bright cells mark the agent's view. Targets and doors are yellow, distractors red and blue, and goals green. In DoorMaze, door factors are purple, and a located door's shade gives its lock: yellow if unknown, green if unlocked, and red if locked. In TreasureHunt, the blue ball is a clue, the yellow box is the hidden treasure, and the brighter disc marks one standard deviation of its `Gaussian`.

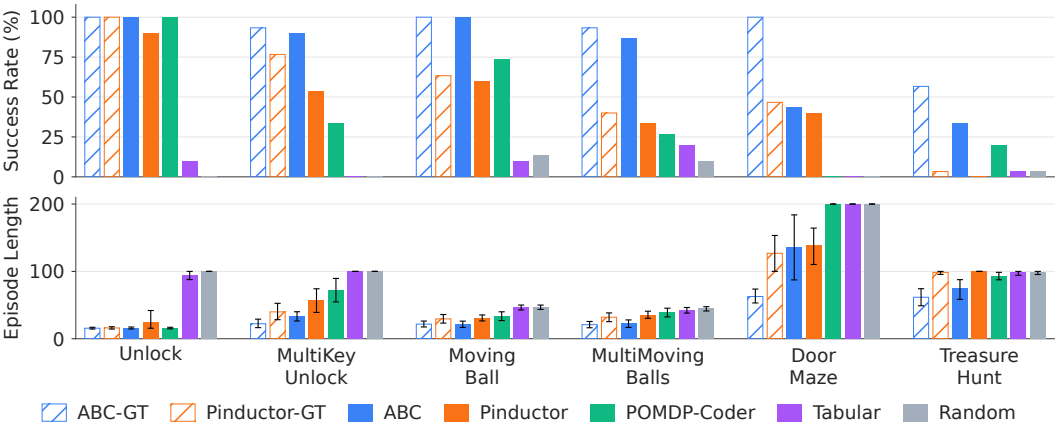


Figure 4: Planning performance on MiniGrid. Success rate (top) and mean episode length (bottom) on six MiniGrid environments. We compare ABC with Pinductor, POMDP-Coder, a tabular planner, and a random policy. ABC and Pinductor also use ground-truth models instead of generated ones. Error bars show 95% percentile confidence intervals of the mean episode length. See Tables 3 and 4 in Appendix G for details.

poses state variables, permitted factor families, and programs that propagate and condition within-particle distributions through a fixed factor library, and hypotheses with different representations are compared by the likelihood of the same observations (Section 2.6). See Appendix E for details.

4 Experiments

We test whether holding uncertainty within particles helps an agent plan when it must track many hidden variables. We use six MiniGrid tasks (Chevalier-Boisvert et al., 2023), shown in Figure 3. The four tasks vary along two axes, the number of hidden objects and whether they move. Unlock tests whether the belief infers where one hidden key lies, and MultiKeyUnlock raises the count to three keys. MovingBall tests whether the belief tracks a ball that moves at random, and MultiMovingBalls raises the count to three. DoorMaze and TreasureHunt involve distributions of other forms: DoorMaze combines a uniform position and a binary lock for each of twelve doors, and TreasureHunt draws the treasure location from a Gaussian and reveals it only through clues. Appendix F gives the rules, and Section 4.4 adds two Crafter tasks with both stationary and moving hidden objects.

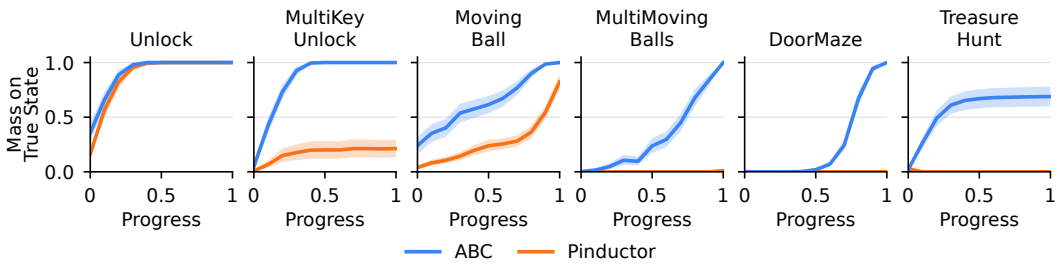


Figure 5: Mass on True State. Probability that ABC and Pinductor, both with ground-truth models, assign to the true complete state over identical action–observation histories. See Table 5 for details.

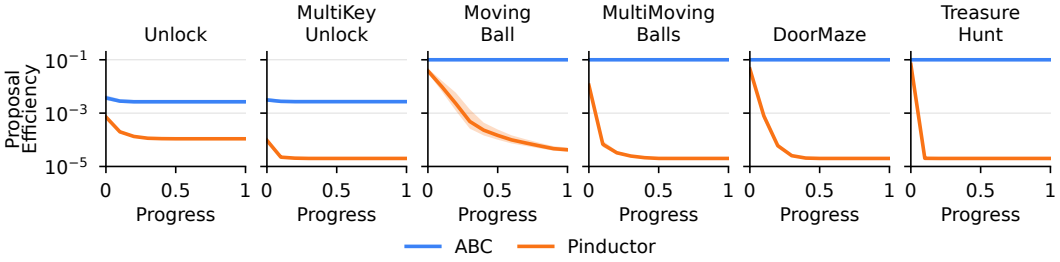


Figure 6: Proposal Efficiency. Inverse of the mean cumulative number of candidate states drawn from I, including initialization; higher values mean fewer proposals. Both filters use ground-truth models; Pinductor's limit of 50,000 applies per episode and ABC's per replenishment call. See Table 6 for details.

4.1 Planning Performance with Abstract Beliefs

We first measure whether retaining more possible states helps the agent reach its goal. We compare ABC with Pinductor and POMDP-Coder, which maintain concrete-particle beliefs, using world models generated by the same LLM from the same offline data, and include tabular planning and a random policy as references. To remove errors from model generation, we also run ABC and Pinductor with human-authored ground-truth models. The concrete-particle methods use a larger search budget; Appendix F.6 lists the remaining differences.

With generated models, ABC succeeds more often than both concrete-particle methods (75.6% versus 46.1% and 42.2%; Figure 4), and the size of this advantage follows the uncertainty the belief must retain. With a single stationary key (Unlock), all three succeed almost always, since a few concrete particles cover the plausible states. The advantage grows when several objects must be tracked at once (MultiKeyUnlock, MultiMovingBalls): each concrete particle commits to one joint placement, so a fixed budget covers fewer of the plausible arrangements, whereas a d-particle keeps each object's uncertainty as a factor. ABC's success drops in DoorMaze and TreasureHunt.

Ground-truth models remove generation errors and raise ABC's success to 100.0% in DoorMaze but only to 56.7% in TreasureHunt, so these errors explain the whole drop in DoorMaze but only part of it in TreasureHunt. ABC's advantage remains (90.6% versus 55.0% for Pinductor with a ground-truth model) and widens most in these two tasks. Both tasks require accumulating indirect evidence, toggle outcomes in DoorMaze and transient clues in TreasureHunt, and we next examine how each belief incorporates the same observations.

4.2 Analyses of Belief Updates

Planning success mixes the belief with the actions chosen from it, so it cannot show whether ABC wins because its belief tracks the hidden state better. To test this, we run only the filters of ABC-GT and Pinductor-GT on identical action–observation histories and compare how much probability each belief keeps on the true state. Both use ground-truth models here and in Section 4.3, so the two filters differ in their belief representation. We measure Mass on True State, the probability $\hat{b}_t(c_t^*)$ that the belief assigns to the true complete state, and the cumulative number of candidate states drawn from I, which measures the cost of regenerating particles.

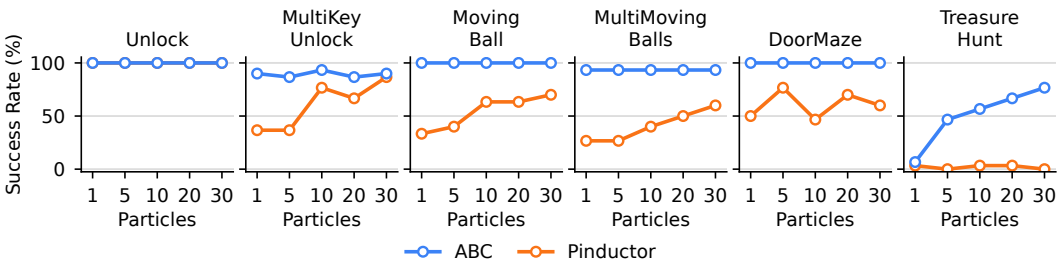


Figure 7: Success rate versus particle count. ABC and Pinductor with ground-truth models and $K \in \{1, 5, 10, 20, 30\}$ particles, 30 episodes per point. The proposal limit grows with K, and each method keeps its search budget from Section 4.1. See Table 7 for details.

ABC concentrates its mass on the true state in every task (Figure 5). Pinductor matches it only when one stationary key is hidden (Unlock) and loses the true state as more objects must be tracked together and as they move, retaining essentially none with three moving balls or twelve doors (MultiMovingBalls, DoorMaze). An observation that contradicts one object's position invalidates a concrete particle, which must place every object correctly at once, but only shrinks the corresponding factor of a d-particle, so ABC keeps accumulating evidence about each object. Factors also let ABC integrate indirect evidence: in TreasureHunt, it conditions the treasure's `Gaussian` on each clue in closed form, updating every candidate location at once, whereas Pinductor scores particles by sampled clues and assigns the true state no mass. Figure 3 shows these updates within single episodes: viewed cells leave each key's factor until the key is seen and its factor collapses to one cell, a ball's factor collapses when the ball is seen and spreads again once it leaves the view, each door's factor narrows from three wall cells to one while toggling fixes its lock, and the treasure's `Gaussian` moves toward the clues and contracts. In every panel after the prior, a single d-particle holds all of these factors; TreasureHunt spreads the treasure over d-particles, and its panels follow one of them.

Drawing more candidates does not close this gap (Figure 6). ABC draws almost no candidates after the first few observations, whereas Pinductor draws tens to thousands of times more and reaches its limit in four tasks. Each new concrete candidate faces the same joint requirement as the particle it replaces, so replenishment does not recover the true state once several variables are hidden. Because proposals replace rejected particles without increasing how many states the belief holds at once, we next increase the particle count together with the proposal limit (Section 4.3).

4.3 Effect of the Particle Budget

To test whether more concrete particles close the gap, we vary the particle count of both filters from 1 to 30 and raise the proposal limit accordingly. Figure 7 shows that ABC with one particle matches or exceeds Pinductor with thirty in every task, and one d-particle already suffices in every task but TreasureHunt. ABC needs few particles because uncertainty stays inside factors, so its updates never split a d-particle and identical d-particles merge. In DoorMaze, for example, its belief is a single d-particle at every decision step. In TreasureHunt, the ground-truth model instead spreads the treasure's `Gaussian` over d-particles as mixture components, so ABC's success rises from 6.7% with one particle to 76.7% with thirty. Pinductor approaches ABC only in MultiKeyUnlock, where the objects are stationary, and a large gap remains with moving objects, locks revealed by interaction, or indirect clues. More concrete particles do not substitute for holding uncertainty inside particles, because each concrete particle still commits every hidden variable to one value. Appendix G.3 reports every setting.

4.4 Planning in Crafter

We then ask whether these gains hold in a richer environment where different kinds of hidden objects appear together. The two Crafter (Hafner, 2021) tasks in Figure 8 hide resources that stay at unknown cells, as the keys do in MultiKeyUnlock, and a zombie that moves out of view like the ball in MovingBall but, unlike the ball, chases and attacks the agent. In Zombie Escape, the agent must cut wood from one of several trees at unknown cells, and in Collect Iron, it must mine iron hidden among many stone cells. In both tasks, the agent sees only its surroundings, never its absolute position, and

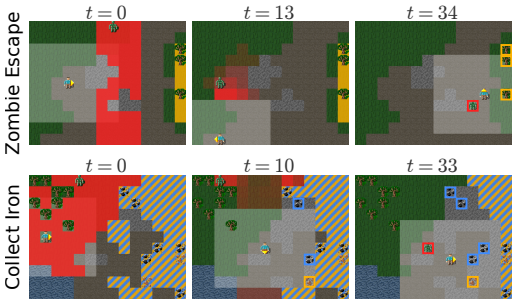


Figure 8: Crafter tasks and the d-particles of ABC. One episode of ABC with the ground-truth model per task; bright cells mark the agent's view. Shading shows the factors of the zombie (red), the target resource, wood or iron (yellow), and coal (blue) beneath the true objects, and outlines mark located objects. Each belief is a single d-particle.

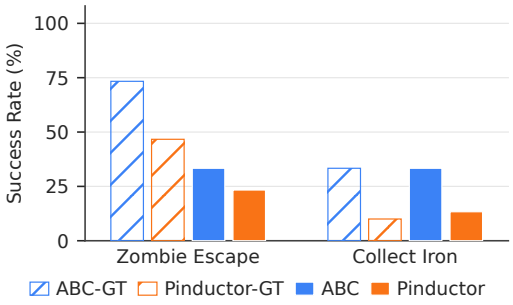


Figure 9: Planning on Crafter. Abstract beliefs keep their advantage with hidden resources and a pursuing zombie, and in Collect Iron even the generated ABC model beats Pinductor with a ground-truth model. See Table 8 for details.

fails if the zombie kills it. We run both methods with ground-truth models and with one generated model per task, and Appendix F gives the full rules.

Abstract beliefs keep their advantage when both kinds of hidden objects must be tracked together: with ground-truth models, ABC succeeds more often than Pinductor in both tasks, as Figure 9 shows. An abstract belief keeps a distribution over the zombie's position while it is out of view and spreads each resource over its candidate cells, whereas a concrete particle must commit to one zombie position and one placement of every resource. Figure 8 shows this for ABC: one d-particle holds the zombie's factor, which spreads while the zombie is out of view and collapses when it is seen, together with resource factors that shrink as the agent views the terrain; Appendix H contrasts both methods. In Collect Iron, the representation matters even more than a correct model: generated ABC models succeed more often than Pinductor with a ground-truth model. In Zombie Escape, however, generated ABC models trail ABC with the ground-truth model by a wide margin, so better model generation could raise success there.

5 Limitations and Conclusion

Limitations. Although abstract beliefs improve planning, LLM-generated models still struggle in the more complex tasks, which suggests that more effective ways of generating and revising world models with LLMs could further improve performance. The factor library currently provides a small set of distribution families, and environments with richer state structure may require additional families. Finally, our evaluation is limited to grid-world environments, and it remains to be seen how well ABC extends to more complex settings, such as continuous states with nonlinear dynamics.

Conclusion. Code world models for partially observable environments have expressed uncertainty only across particles, so each particle commits every hidden variable to one value, and a fixed budget covers fewer plausible states as hidden variables accumulate. We presented the Abstract Belief Coder (ABC), whose distributional particles (d-particles) also hold uncertainty *within* a particle: a view that rules out one object's position shrinks that object's factor instead of discarding the particle and what it has inferred about the other objects. An LLM chooses the factor family of each variable and writes the programs that update them through a shared Factor Library. In MiniGrid, ABC succeeds more often than concrete-particle beliefs with both generated and ground-truth models, and a single d-particle matches or exceeds thirty concrete particles, so the gain comes from the belief representation rather than from model generation or particle count; ABC also leads on two Crafter tasks.

REFERENCES

Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and Michael Carbin. Semi-symbolic inference for efficient streaming probabilistic programming. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2), 2022.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, 20(28):1–6, 2019.

Craig Boutilier and David Poole. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the 13th National Conference on Artificial Intelligence*, pp. 1168–1175, 1996.

Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

Rong Chen and Jun S. Liu. Mixture Kalman filters. *Journal of the Royal Statistical Society: Series B*, 62(3):493–508, 2000.

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.

Aidan Curtis, Hao Tang, Thiago Veloso, Kevin Ellis, Joshua B. Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. LLM-guided probabilistic program induction for POMDP model estimation. In *Proceedings of the 9th Annual Conference on Robot Learning*, 2025.

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen: A general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 221–236, 2019.

Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. Generating code world models with large language models guided by Monte Carlo tree search. In *Advances in Neural Information Processing Systems*, volume 37, pp. 60429–60474, 2024.

Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. Lifted first-order probabilistic inference. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pp. 1319–1325, 2005.

Carlos Diuk, Andre Cohen, and Michael L. Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th International Conference on Machine Learning*, pp. 240–247, 2008.

Arnaud Doucet, Nando de Freitas, Kevin Murphy, and Stuart Russell. Rao-Blackwellised particle filtering for dynamic Bayesian networks. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pp. 176–183, 2000.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, pp. 220–229, 2008.

Nishad Gothoskar, Matin Ghavami, Eric Li, Aidan Curtis, Michael Noseworthy, Karen Chung, Brian Patton, William T. Freeman, Joshua B. Tenenbaum, Mirko Klukas, and Vikash K. Mansinghka. Bayes3D: Fast learning and inference in structured generative models of 3D objects and scenes. *arXiv preprint arXiv:2312.08715*, 2023.

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 *Advances in Neural Information Processing Systems*, volume 36, 2023.

David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems*, volume 31, 2018.

Danijar Hafner. Benchmarking the spectrum of agent capabilities. *arXiv preprint arXiv:2109.06780*, 2021.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. In *International Conference on Learning Representations*, 2020.

Ivy He, Stefanie Tellex, and Jason Xinyu Liu. LEGS-POMDP: Language and gesture-guided object search in partially observable environments. In *Proceedings of the ACM/IEEE International Conference on Human-Robot Interaction*, 2026. doi: 10.1145/3757279.3785585.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101:99–134, 1998. URL `https://api.semanticscholar.org/CorpusID:5613003`.

Zaid Khan, Archiki Prasad, Elias Stengel-Eskin, Jaemin Cho, and Mohit Bansal. One life to learn: Inferring symbolic world models for stochastic environments from unguided exploration. In *International Conference on Learning Representations*, 2026.

Tejas D. Kulkarni, Pushmeet Kohli, Joshua B. Tenenbaum, and Vikash Mansinghka. Picture: A probabilistic programming language for scene perception. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4390–4399, 2015.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. LLM+P: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023.

Panagiotis Lymperopoulos, Abhiramon Rajasekharan, Ian Berlot-Attwell, Stéphane Aroca-Ouellette, and Kaheer Suleman. CASSANDRA: Programmatic and probabilistic learning and inference for stochastic world modeling. *arXiv preprint arXiv:2601.18620*, 2026.

Rajesh Mangannavar, Alan Fern, and Prasad Tadepalli. Hierarchical object-oriented POMDP planning for object rearrangement. *arXiv preprint arXiv:2412.01348*, 2024.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic models with unknown objects. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pp. 1352–1359, 2005.

Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pp. 593–598, 2002.

Kevin Murphy and Stuart Russell. Rao-Blackwellised particle filtering for dynamic Bayesian networks. In Arnaud Doucet, Nando de Freitas, and Neil Gordon (eds.), *Sequential Monte Carlo Methods in Practice*, pp. 499–515. Springer, 2001.

Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. Delayed sampling and automatic Rao-Blackwellization of probabilistic programs. In *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics*, pp. 1037–1046, 2018.

Top Piriyakulkij, Yichao Liang, Hao Tang, Adrian Weller, Marta Kryven, and Kevin Ellis. PoE-World: Compositional world modeling with products of programmatic experts. *Advances in Neural Information Processing Systems*, 38:26609–26638, 2026.

David Poole. First-order probabilistic inference. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pp. 985–991, 2003.

Jorma Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

Valentin Six, Fabian Panse, Mathieu Fajeau, Lancelot Da Costa, Mrinank Sharma, Alfonso Amayuelas, Tim Z. Xiao, David Hyland, Philipp Hennig, and Bernhard Schölkopf. Learning POMDP world models from observations with language-model priors. *arXiv preprint arXiv:2605.13740*, 2026.

Hao Tang, Darren Key, and Kevin Ellis. WorldCoder, a model-based LLM agent: Building world models by writing code and interacting with the environment. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Wenjing Tang, Xinyu He, Yongxi Huang, Yunxiao Xiao, Cewu Lu, and Panpan Cai. Tru-POMDP: Task planning under uncertainty via tree of hypotheses and open-ended POMDPs. *Advances in Neural Information Processing Systems*, 38:55367–55396, 2026.

Arthur Wandzel, Yoonseon Oh, Michael Fishman, Nishanth Kumar, Lawson L. S. Wong, and Stefanie Tellex. Multi-object search using object-oriented POMDPs. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 7194–7200, 2019.

Lionel Wong, Gabriel Grand, Alexander K. Lew, Noah D. Goodman, Vikash K. Mansinghka, Jacob Andreas, and Joshua B. Tenenbaum. From word models to world models: Translating from natural language to the probabilistic language of thought. *arXiv preprint arXiv:2306.12672*, 2023.

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*, pp. 1024–1032, 2014.

Takuma Yoneda, Jiading Fang, Peng Li, Huanyu Zhang, Tianchong Jiang, Shengjie Lin, Ben Picker, David Yunis, Hongyuan Mei, and Matthew R Walter. Statler: State-maintaining language models for embodied reasoning. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 15083–15091. IEEE, 2024.

A USAGE OF LARGE LANGUAGE MODELS

The authors used large language models to refine the writing, improve grammar, and enhance the readability of this manuscript, and critically reviewed, verified, and edited all content; they assume full responsibility for the final version. Beyond writing assistance, LLMs are part of the evaluated methods: ABC and the LLM baselines generate world models with `gpt-5.6-luna` through the Codex CLI on MiniGrid, and ABC and Pinductor with `gpt-5.6-sol` on Crafter (Section 2.6 and Appendices D and F.6).

B METHOD DETAILS

B.1 MODEL INTERFACES AND THE BOUNDARY WITH THE LIBRARY

The four programs of Section 2.1 have the following signatures:

```
def initial(empty_state) -> z:
    ...  # sample an abstract state from the prior

def transition(z, a) -> z_bar:
    ...  # predict an abstract successor

def correct(z_bar, o) -> (z, l):
    ...  # condition the predicted state on the actual observation

def reward(source, a, target) -> (r, terminal):
    ...  # compute reward and termination status
```

The representation ρ is written as type declarations such as those in Section 2.2. I chooses the initial factor of each state variable, and T and O update them.

Division of labor. The world model supplies kernels and likelihoods for a single value, and the Factor Library iterates over the factor's domain; this iteration replaces enumerating each value as a separate particle. For a stationary key that is not in view, for example, O supplies a likelihood that is zero on the cells the view covers, and `condition` multiplies the key factor by it, renormalizes, and returns the evidence.

Kernels do not read the belief. A transition kernel $\tau(x' \mid x)$ depends only on environment constraints such as walls, never on the current factor parameters, so a cell excluded by one view can regain probability in prediction.

B.2 FILTERING DETAILS

The agent runs Algorithm 1 at episode start with $P = \emptyset$ and after every action. The history $h = (o_0, [(a_0, o_1), \ldots, (a_{t-1}, o_t)])$ includes the current observation, and I uses only the known environment information $z_\emptyset$.

Adding to $W(z)$ merges identical abstract states. Algorithm 2 shows how the programs T and O that Algorithm 1 calls are structured in the MiniGrid tasks. When the observation likelihood decomposes by factor, the particle likelihood ℓ is the product of the evidence $e_j = \int \eta_j(x) p_j(x)\, dx$ that `condition` returns for each factor j; if occupancy or other rules couple factors, O must account for their joint probability. An omitted observation leaves the prediction unchanged with $\ell = 1$, and $o_0 = \text{None}$ denotes an omitted initial observation.

B.3 PLANNING DETAILS

Representative placement. T places each non-`Dirac` position factor at one representative cell, close to its marginal mean and distinct from the cells of other factors, and evaluates interactions such as pickup and toggle there. The placement is never stored: a successful pickup removes the key factor and updates the carrying state, a failed one leaves the factor unchanged, and the remaining

Algorithm 1 Particle initialization and belief update

Require: Previous belief P, action a, observation o, history h; models I, T, O
Require: Known information $z_\emptyset$, target K, prior-draw limit $J_{\max}$

Update the current particles
$W(z) \leftarrow 0$ for all z ▷ Unnormalized weight assigned to state z
for all particles (z, w) in P **do** ▷ State hypothesis z and its previous weight w
  $\bar{z} \leftarrow T(z, a)$ ▷ Predict the effect of the executed action
  $(z', \ell) \leftarrow O(\bar{z}, o)$ ▷ Posterior state and observation likelihood
  if $\ell > 0$ **then**
    $W(z') \leftarrow W(z') + w\ell$ ▷ Reweight; sum weights of identical states
  end if
end for

Draw initial particles and replay the history if needed
if $\sum_z W(z) = 0$ **then** ▷ Initialize, or recover after all particles are rejected
  $n \leftarrow 0$ ▷ Accepted prior proposals
  for $j = 1, \ldots, J_{\max}$ **do**
    $z \sim I(z_\emptyset)$ ▷ Draw a fresh initial-state hypothesis
    $(z, w) \leftarrow O(z, o_0)$ ▷ Initial observation sets the proposal weight
    for all (a_i, o_{i+1}) in h, in time order **do** ▷ Replay past interactions on this proposal
      if $w = 0$ **then**
        break ▷ An earlier observation ruled out this proposal
      end if
      $\bar{z} \leftarrow T(z, a_i)$
      $(z, \ell) \leftarrow O(\bar{z}, o_{i+1})$
      $w \leftarrow w\ell$ ▷ Accumulate likelihood over the full history
    end for
    if $w > 0$ **then** ▷ Keep proposals consistent with every observation
      $W(z) \leftarrow W(z) + w$
      $n \leftarrow n + 1$
    end if
    if $n = K$ **then**
      break ▷ Enough prior proposals have survived replay
    end if
  end for
end if

Normalize the resulting belief
$Z \leftarrow \sum_z W(z)$ ▷ Total surviving weight
if $Z = 0$ **then**
  stop with initialization failure
end if
return $\{(z, W(z)/Z) : W(z) > 0\}$ ▷ Divide by Z so the weights sum to one

factors are then propagated in the environment's motion order. R likewise takes abstract source and successor states.

Successors. Each successor inherits its parent particle's weight, and successors with identical states and termination flags are merged. Terminated particles are absorbing: they are not expanded again and contribute their reward only once. If only some particles terminate, the rest continue without renormalizing the surviving mass.

Search. The open-loop planner runs best-first search on the accumulated cost, which expanding action a from belief b increases by $c_a - \bar{r}(b, a)$, where $\bar{r}$ is the weighted predicted reward and c_a a constant action cost; no heuristic estimates the remaining distance to the goal. A new path to an already reached belief is discarded if an existing path has no greater cost and no greater depth, and paths tied in both keep their first actions together. When the search budget is exhausted, the planner compares unexpanded frontier nodes, fully terminated candidates, and candidates at the action limit,

Algorithm 2 Structure of the programs T and O. Only $\mathrm{Propagate}$ and $\mathrm{Condition}$ are Factor Library calls; every other line is an environment rule that the hypothesis writes. O returns $\ell = 0$ for an observation that no variable explains, such as a wall where $\bar{z}$ has none; this check is omitted.

function $T(z, a)$ ▷ Abstract state z and executed action a
  Evaluate the action at a representative placement
  for all position factors F_j in z **do**
    $v_j \leftarrow$ supported cell of F_j nearest its marginal mean ▷ Distinct from other objects' cells
  end for
  $y \leftarrow$ outcome of a with each object at v_j ▷ Turn, move, pick up, toggle
  Propagate every factor
  $\bar{z} \leftarrow z$ ▷ Placement v is not stored
  for all factors F_j in $\bar{z}$, in the environment's motion order **do**
    $\tau_j \leftarrow$ kernel of x_j given y and the motion of x_j ▷ Never reads F_j's parameters
    $F_j \leftarrow \mathrm{Propagate}(F_j, \tau_j)$ ▷ Predicted factor
  end for
  remove the factor of an object picked up in y from $\bar{z}$
  return $\bar{z}$
end function

function $O(\bar{z}, o)$ ▷ Predicted abstract state $\bar{z}$ and actual observation o
  $V \leftarrow$ cells in view, from the agent's pose and the walls in $\bar{z}$
  $z \leftarrow \bar{z}$ ▷ Posterior abstract state
  $\ell \leftarrow 1$ ▷ Likelihood of o under $\bar{z}$
  for all factors F_j in $\bar{z}$ **do**
    if o shows the object of x_j at a cell $s \in V$ **then**
      $\eta_j(u) \leftarrow \mathbf{1}[u = s]$ ▷ Only $u = s$ explains the sighting
    else
      $\eta_j(u) \leftarrow \mathbf{1}[u \notin V]$ ▷ The view rules out its cells
    end if
    $(F_j', e_j) \leftarrow \mathrm{Condition}(F_j, \eta_j)$ ▷ Posterior factor and its evidence
    replace F_j with F_j' in z
    $\ell \leftarrow \ell\, e_j$ ▷ Particle likelihood
  end for
  return (z, ℓ)
end function

excluding the root and falling back to the expanded nodes if none exist. It chooses uniformly among the distinct first actions that lead to minimum-cost candidates, or among all valid actions if there is none. The previous search destination also remains a candidate if the actual posterior matches the belief predicted for the last action and the current search reaches it again, with its cost recomputed; this keeps the agent moving toward a chosen frontier in reward-free regions.

C THE FACTOR LIBRARY

Table 1 lists the families of the Factor Library and the kernels and likelihoods that each accepts in the two operations of Equation 3.

Table 1: Factor families of the Factor Library. Propagation with a kernel in $\mathcal{K}_f$ and conditioning on a likelihood in $\mathcal{L}_f$ keep the distribution in the family; kernels must also preserve the domain.

Family	Domain $\mathcal{U}_f$	Parameter q	Kernel class $\mathcal{K}_f$	Likelihood class $\mathcal{L}_f$
`Dirac`	Any set	Value v	Deterministic only	Any nonnegative η
`Uniform`	Finite set	Candidate set S	Identity only	0/1 indicators
`Categorical`	Finite set	Probability vector	Any Markov kernel	Any nonnegative η
`Bernoulli`	$\{0, 1\}$	$p = P(B = 1)$	Any Markov kernel	Any nonnegative η
`Gaussian`	$\mathbb{R}^d$	Mean μ, covariance Σ	Affine-Gaussian	Linear-Gaussian

C.1 WHAT THE TWO OPERATIONS GUARANTEE

A factor stores its family, its domain, and its parameter. Each family accepts only kernels under which the prediction stays in the family (its kernel class $\mathcal{K}_f$) and likelihoods under which the posterior stays in the family (its likelihood class $\mathcal{L}_f$), and the operation rejects any other update instead of approximating it. Propagation also rejects a kernel that leaves the domain, and conditioning rejects likelihood values that are not finite and nonnegative.

The evidence ℓ is the probability that the factor assigned to the observation before the update, and O multiplies the evidence of all factors into the particle's likelihood. If $\ell = 0$, `condition` leaves the parameter unchanged, and filtering removes the particle. For `Gaussian`, ℓ is a density, whose value depends on the units of the observation.

C.2 THE FIVE FAMILIES

Dirac. `Dirac` stores a single value v, such as the agent's cell. It accepts deterministic kernels, which move v to one successor value, and every nonnegative likelihood η; conditioning leaves v unchanged and returns $\ell = \eta(v)$, which is zero when the observation contradicts v. When every factor is `Dirac`, the filter reduces to a concrete-particle filter.

Uniform. `Uniform` stores the set S of candidate values, all equally likely. It accepts only the identity kernel and likelihoods that are 1 on a set A and 0 elsewhere, and conditioning reduces to

$$S' = S \cap A, \qquad \ell = \frac{|S'|}{|S|}. \tag{C.1}$$

Categorical. `Categorical` stores one probability per value, $q_i = P(X = u_i)$, and accepts every Markov kernel τ that preserves the domain and every nonnegative likelihood η:

$$\bar{q}_j = \sum_i q_i \, \tau(u_j \mid u_i), \qquad \ell = \sum_j \bar{q}_j \, \eta(u_j), \qquad q'_j = \frac{\bar{q}_j \, \eta(u_j)}{\ell}. \tag{C.2}$$

`Uniform` is the special case of a stationary variable observed only through inclusion and exclusion; as a separate family, it rejects updates outside this case.

Bernoulli. `Bernoulli` stores $p = P(B = 1)$, such as the probability that a DoorMaze door is locked. Locks never change, so only observations update p, and a closed door leaves p unchanged with $\ell = 1$. After the agent toggles the closed door ahead, a door that stays closed sets $p = 1$ with evidence p, and a door that opens sets $p = 0$ with evidence $1 - p$. `Bernoulli` is a two-valued `Categorical` with a fixed domain, so propagation rejects a kernel that returns any other value without the world model declaring the domain.

Gaussian. `Gaussian` stores a mean and a covariance and accepts exactly the transitions $X' = AX + b + \varepsilon$ and observations $y = HX + \nu$ with Gaussian noise ε and ν, which it propagates and conditions with the Kalman-filter prediction and update; the evidence is the predictive density of y. It rejects nonlinear transitions and non-Gaussian likelihoods rather than linearizing them.

D REPRESENTATION ACQUISITION DETAILS

Each call of Section 2.6 starts a new Codex conversation in a workspace that contains the instructions of Prompt 1, the environment description and goal of Appendix F, the request of the call, and the source files of the predicate state interface and the Factor Library. The agent writes I, T, O, R to a single Python module. The state interface stores each object of an abstract state, such as the agent, a ball, or a room, as a predicate, and a predicate budget bounds their number. A candidate whose programs raise an error during scoring, including an update that the Factor Library rejects, is discarded. The predicate count m_h in Equation 7 is summed over the distinct d-particles of the belief and averaged over the initial belief and every posterior along h. Scoring runs the filter without reinitialization, whose mass depends on when drawing stops; the check of a drawn candidate allows reinitialization, so that a failure does not merely reflect unlucky initial d-particles. Appendix F.6 lists the generation budgets.

```
Read task/request.md, task/environment.md, task/goal.md and the API files.
Implement the requested functions in model.py. The evaluator supplies the documented
API namespace. Do not add imports, classes, or top-level execution to model.py.
Do not use while statements. Use bounded loops and handle exhaustion.
Preserve colors and carrying in observations; do not add absolute agent position.
The request contains only the examples selected by the method.
Follow the I/T/O/R authoring contract in abcwm/wm/abcwm/offline_api.md
for initial generation and every revision. The ban on observed absolute
position does not forbid inferred latent agent poses in the model.
Each initial call returns ONE sampled PredicateState, not a dict or
an enumeration of poses. The runtime collects the prior proposals;
keep uncertain object locations as factors inside each proposal.
mean_distinct_assignment is supplied without imports; its source and
generic arithmetic dependency are included with the API files.
```

Prompt 1: Workspace instructions (`AGENTS.md`) for every ABC call.

Generation prompt. Prompt 2 shows the request of a generation call. The authoring reference states the I/T/O/R contract and the Factor Library interface of Appendix C, and its five training episodes include at least one success and one failure or timeout. A call that extends a candidate instead receives three episodes, its code, and its training scores.

```
<authoring reference>
ENVIRONMENT
<environment description>
GOAL
<goal>
INITIAL CANVAS
PredicateState(width=11, height=6, predicates=(), predicate_budget=64)
OBSERVED DATA
<five offline episodes (three when extending a candidate), each with its
 observations, actions, rewards, and termination flags>
CURRENT CODE (only when extending a candidate)
<the candidate's four functions>
TRAIN FEEDBACK
<score, fit, predicate penalty, and per-episode diagnostics on training episodes>

Explain the errors briefly, then return all four functions in one Python code block.
```

Prompt 2: Generation prompt. Angle brackets mark inserted content; the initial state is shown for `Unlock`.

Repair prompt. When the drawn candidate cannot explain an offline trajectory even with reinitialization, the next call uses the repair prompt (Prompt 3), which adds the candidate's code and the earliest failing prefix of that trajectory. The agent sees only the latest candidate and its failure.

```
<authoring reference>
ENVIRONMENT / GOAL / INITIAL CANVAS
CURRENT CODE
<the selected candidate's four functions>
REPAIR FEEDBACK
OFFLINE zero-evidence REFINEMENT
The selected candidate cannot explain a public offline history within the prior proposal
budget. This is a finite-budget diagnosis, not proof of zero model support. Repair the
current code using only these public offline observations. Preserve previously compatible
histories; all offline histories will be rechecked. Do not make O accept every observation,
invent evidence, or read hidden state. Keep one sampled pose per I call and the I/T/O/R
authoring contract.
```

```
OFFLINE RECOVERY SUMMARY
<status and failed step of every offline episode>
OBSERVATION TABLE (repr, indexed by observation_id)
<distinct observations in the failing prefix>
FAILED OFFLINE HISTORY PREFIX
<(t, action, observation_id, reward, terminated) up to the failed step>
FAILURE DIAGNOSTICS (state excerpt may be truncated)
<failed step, evidence, prior draws, error, and an excerpt of the predicted belief>
Return a complete replacement of all four functions in exactly one Python code block. Do
not return a patch, isolated snippets, or top-level executable statements.

task/request.md:
Repair the existing complete implementation in model.py. Use only the public offline
failure below; do not infer or request evaluation data. Preserve the four-function I/T/O/R
contract and replace model.py with the complete corrected implementation.
<REPAIR FEEDBACK above>
```

Prompt 3: Repair prompt. The first four blocks are those of Prompt 2.

E Extended Related Work

Code world models and LLM-induced symbolic models. Language models generate executable transition and reward models refined against interaction data (Tang et al., 2024; Dainese et al., 2024), translate task descriptions and dynamics into planning-language specifications (Guan et al., 2023; Liu et al., 2023), or synthesize probabilistic programs from language (Wong et al., 2023). POMDP-Coder (Curtis et al., 2025) learns probabilistic POMDP components and tracks beliefs with particle filtering, and Pinductor (Six et al., 2026) learns from observation–action trajectories and scores candidate models by a particle-filtered likelihood. Both use particles that specify complete states, whereas a d-particle holds distributions that the generated programs propagate and condition.

Compositional and hybrid probabilistic world models. PoE-World (Piriyakulkij et al., 2026) combines programmatic experts through an exponentially weighted product and handles partial observability by conditioning on action–observation histories rather than a latent-state belief. ONELIFE (Khan et al., 2026) fits the weights of conditionally activated programmatic laws to observed state transitions in Crafter-OO, and CASSANDRA (Lymperopoulos et al., 2026) combines generated code for deterministic dynamics with a learned Bayesian network for dependent stochastic variables. These methods compose predictive mechanisms, whereas ABC retains uncertainty about the current hidden state, which even a stationary object with deterministic dynamics can require.

Particle filtering with analytic marginalization. Rao–Blackwellized particle filters sample one subset of variables while retaining conditional distributions over the remainder (Doucet et al., 2000; Chen & Liu, 2000; Murphy & Russell, 2001); FastSLAM, for example, samples robot trajectories and maintains landmark posteriors conditional on each trajectory (Montemerlo et al., 2002). When its factors stay independent and every factor update is exact, ABC is an instance of this principle (Section 2.4), but it also searches for the state variables, their factor families, and the programs that update them instead of taking the generative model as given.

Probabilistic programming and automatic marginalization. Probabilistic programming languages provide reusable inference for structured generative models (Milch et al., 2005; Goodman et al., 2008; Wood et al., 2014; Cusumano-Towner et al., 2019; Bingham et al., 2019). Delayed sampling (Murray et al., 2018) and semi-symbolic inference (Atkinson et al., 2022) compute analytically where conjugacy permits and sample otherwise, and Picture and Bayes3D (Kulkarni et al., 2015; Gothoskar et al., 2023) infer scenes with structured generative programs. These systems execute a supplied model program, whereas ABC also searches for the program and its state representation.

Structured state representations for sequential decision making. Factored MDP and POMDP formulations exploit conditional independence (Boutilier & Poole, 1996; Boutilier et al., 1999), and object-oriented formulations organize states and observations around objects (Diuk et al., 2008; Wandzel et al., 2019). OO-POMCP maintains independent object distributions for multi-object search (Wandzel et al., 2019), HOO-POMDP extends object-oriented belief maintenance to rearrangement with an abstract planner and low-level policies (Mangannavar et al., 2024), and lifted inference exploits symmetry among interchangeable individuals (Poole, 2003; de Salvo Braz et al., 2005). ABC shares structured beliefs but generates the state vocabulary and factor families as part of each hypothesis.

Belief tracking in language-model agents. Statler maintains an explicit textual world state (Yoneda et al., 2024), which does not assign probabilities to alternative states. Tru-POMDP converts a tree of hypotheses into complete-state particles and combines Bayesian filtering with LLM-generated particle supplementation (Tang et al., 2026), and LEGS-POMDP incorporates language, gesture, and visual evidence into a probabilistic observation model for object search (He et al., 2026). Neural world models learn recurrent or latent predictive representations from experience (Ha & Schmidhuber, 2018; Hafner et al., 2020). ABC instead uses the LLM to propose named state variables and executable dynamics and keeps unresolved uncertainty in explicit factors.

F Environments and Experimental Setting

F.1 Overview

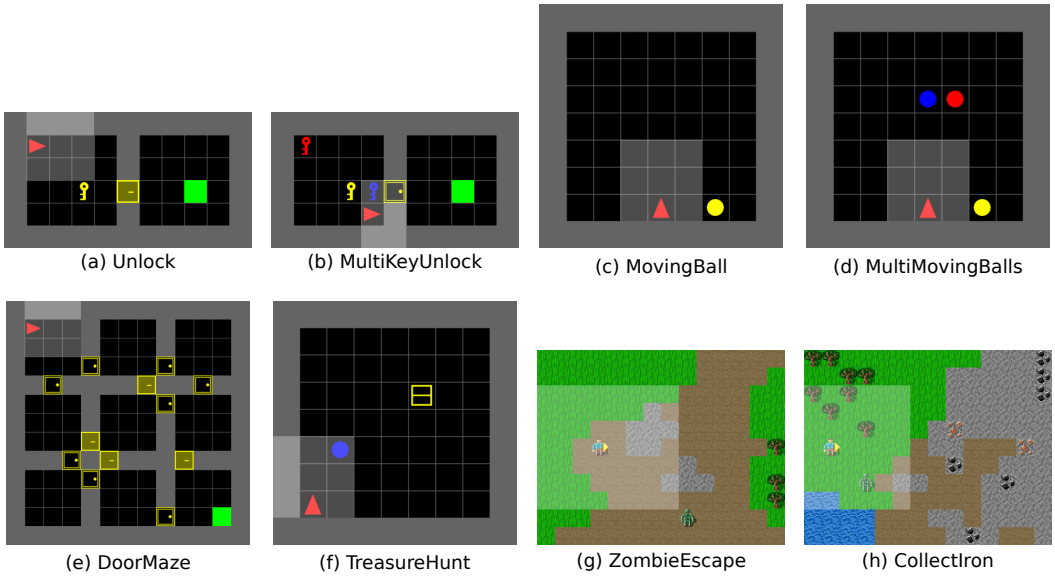


Figure 10: Evaluation environments. (a)–(f) MiniGrid and (g)–(h) Crafter. Bright cells mark the agent's current view. In (f), the blue ball is a clue and the yellow box marks the treasure, which never appears in observations.

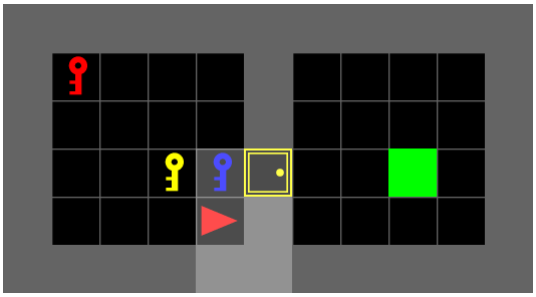
Figure 10 shows the six MiniGrid environments and the two Crafter tasks. Appendices F.3 and F.4 give their rules and the descriptions supplied to the model generators.

F.2 Observation Format

A MiniGrid observation contains the visible cells and objects of an agent-centered 3×3 view, which extends two cells forward and one to either side with the agent at the rear center, together with the agent's orientation and inventory. Figure 11 shows an example.

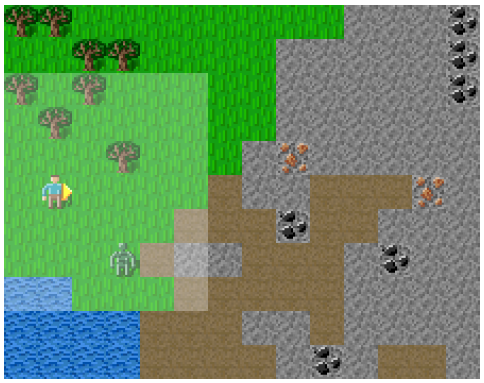
Crafter observations. The Crafter tasks use a 9×7 view centered on the player, matching the terrain area of the native Crafter local view. It is aligned with the world axes and covers four cells to either side horizontally and three above and below. Each cell is a symbol string such as `grass`, `path`, `water`, `tree`, `stone`, `coal`, `iron`, or `zombie`; cells outside the 14×11 map appear as `boundary`, and the sprite of the player or a zombie replaces the terrain beneath it. The observation also includes the player's absolute facing vector and the observable inventory entries, health and wood in `Zombie Escape` and health, iron, and wood in `Collect Iron`. It omits the player's absolute position, the full map, the zombie's health and attack cooldown, and hidden inventory such as the stone pickaxe in `Collect Iron`. Figure 12 shows an example.

Descriptions supplied to the model generators. Figures 13–20 show the description and goal supplied to the model generators for each environment, and the text below states the rules and modeling choices that the description leaves implicit. Within a task, the MiniGrid description and goal are shared by all compared methods, and the Crafter ones by ABC and Pinductor, the only methods evaluated on Crafter.



```
MinigridObservation(agent_dir=0, carrying=None, image=[
    [Empty(), Empty(), Empty()],
    [Door(color='yellow', state='closed'), Wall(), Wall()],
    [Key(color='blue'), Agent(), Wall()],
])
```

Figure 11: Partial observation in MultiKeyUnlock. Top: an enlarged view of the scene in Figure 10(b), where the red triangle is the agent facing right and the bright region is its view; the complete map is not provided to the agent. Bottom: the corresponding observation. The center of the last row of `image` is the agent's own cell, and higher rows are farther forward, so the blue key and the closed yellow door appear, whereas the yellow and red keys and the green goal outside the view are absent.



```
CrafterObservation(facing=(1, 0),
  inventory=(('health', 9), ('iron', 0), ('wood', 0)),
  image=(
    ('boundary', 'boundary', 'boundary', 'boundary', 'boundary', 'boundary', 'boundary'),
    ('boundary', 'boundary', 'boundary', 'boundary', 'boundary', 'boundary', 'boundary'),
    ('boundary', 'boundary', 'boundary', 'boundary', 'boundary', 'boundary', 'boundary'),
    ('tree', 'grass', 'grass', 'grass', 'grass', 'grass', 'water'),
    ('grass', 'tree', 'grass', 'player-right', 'grass', 'grass', 'water'),
    ('tree', 'grass', 'grass', 'grass', 'grass', 'grass', 'grass'),
    ('grass', 'grass', 'tree', 'grass', 'grass', 'zombie', 'grass'),
    ('grass', 'grass', 'grass', 'grass', 'grass', 'path', 'grass'),
    ('grass', 'grass', 'grass', 'grass', 'path', 'stone', 'path'),
  ))
```

Figure 12: Partial observation in Collect Iron. Top: the scene in Figure 10(h) with the player's 9×7 view highlighted; the player faces right, the zombie lies inside the view, and the coal and iron cells on the right lie outside it. The complete map is not provided to the agent. Bottom: the corresponding observation. `image[i][j]` is the cell at offset $(i-4,\ j-3)$ from the player, with the first coordinate increasing to the right and the second downward, so each tuple is one column of the view and the first three columns lie outside the map.

F.3 MiniGrid Environments

F.3.1 Unlock

```
Grid world with boundary walls and a locked door separating the map
into two sections. A yellow key is placed in the accessible section,
and the goal is located behind the matching yellow locked door. Agent
starts in the initial section with random position and direction. The
room layout, door, goal, and object color are fixed; only the agent
pose and key position VARY between episodes.
```

```
Find the yellow key, open the door, and reach the goal in the right
room.
```

Figure 13: `Unlock`: description and goal supplied to the model generators. Top: environment description. Bottom: goal.

Episodes time out after 100 actions.

F.3.2 MultiKeyUnlock

```
A fixed 11x6 grid world has two rooms separated by a yellow locked
door. The accessible left room contains three keys: one yellow key
matching the door, plus red and blue distractor keys. The goal is in
the right room behind the door. The agent starts with random position
and direction in the left room. The room size and number of
distinct-color keys are configurable; only the agent pose and key
positions VARY between episodes.
```

```
Use the yellow key, not the red or blue one, to open the door and
reach the goal.
```

Figure 14: `MultiKeyUnlock`: description and goal supplied to the model generators. Top: environment description. Bottom: goal.

The map is that of `Unlock`, and all three keys are stationary. Episodes time out after 100 actions.

F.3.3 DoorMaze

```
A fixed 13x13 grid contains nine rooms in a three-by-three
arrangement, each with a 3x3 interior, and twelve yellow doors. Each
shared wall has one door sampled uniformly from its three interior
cells at reset. Every door starts closed. A closed door may be locked,
but its appearance does not reveal this. Toggle opens or closes an
unlocked door; a locked door stays closed. There are no keys. Locks
remain fixed throughout an episode and VARY between episodes. A route
to the goal always exists. Use movement, turning and toggle to reach
the green goal in the bottom-right room.
```

```
Explore the nine rooms, infer hidden door locks from toggle outcomes,
and reach the goal without keys.
```

Figure 15: `DoorMaze`: description and goal supplied to the model generators. Top: environment description. Bottom: goal.

Room boundaries lie at $0, 4, 8, 12$ on both axes, and each door occupies one of the three nonendpoint cells of its shared wall. To guarantee a route, generation first selects a backbone of eight unlocked doors connecting all nine rooms by randomized depth-first search from a uniformly chosen room, and then locks each remaining door independently with probability 0.5. Episodes time out after 200 actions.

F.3.4 MOVINGBALL

```
9x9 grid with boundary walls and an unobstructed 7x7 interior. There
are no internal walls or restricted movement regions. The agent starts
at (4, 7) facing up. The yellow ball starts uniformly in one of the 48
interior cells other than the agent's starting cell. After every
non-terminal action the ball attempts one uniformly random cardinal
step. It stays put if the destination is outside the interior or
occupied by the agent or an object. It may move anywhere in the
interior. Picking up yellow gives reward +1 and immediately ends the
episode, before any further ball motion. The initial ball position and
random walk VARY.
```

```
Catch the moving yellow ball by picking it up.
```

Figure 16: `MovingBall`: description and goal supplied to the model generators. Top: environment description. Bottom: goal.

On every step other than a successful pickup, including one that reaches the 50-action time limit, the ball moves before the next observation.

F.3.5 MULTIMOVINGBALLS

```
9x9 grid with boundary walls and an unobstructed 7x7 interior. There
are no internal walls or restricted movement regions. The agent starts
at (4, 7) facing up. Yellow, red and blue balls start uniformly
without replacement in the 48 interior cells other than the agent's
starting cell. After every non-terminal action the balls attempt one
uniformly random cardinal step each, sequentially in yellow, red, blue
order, using the occupancy after earlier moves. A ball stays put if
the destination is outside the interior or occupied by the agent or an
object. All balls may move anywhere in the interior. Yellow pickup
gives reward +1; red or blue pickup gives reward -1. Any ball pickup
immediately ends the episode, before further ball motion. Initial ball
positions and random walks VARY.
```

```
Pick up the yellow ball while avoiding the moving red and blue balls.
```

Figure 17: `MultiMovingBalls`: description and goal supplied to the model generators. Top: environment description. Bottom: goal.

Episodes time out after 50 actions. Because each ball checks collisions against the layout left by earlier moves, the balls' motions are not independent.

F.3.6 TREASUREHUNT

The treasure never appears in observations, even when its cell is visible, and neither it nor a clue blocks movement. Each new clue replaces the previous one and is shown as a blue ball only if it falls inside the current view and not on the treasure cell. A failed pickup draws its clue only from visible interior cells other than the agent's and the treasure's. A successful pickup records a carried yellow box, and episodes without success time out after 100 actions.

```
A fixed 9x9 room has a seven-by-seven interior. The agent starts at
(1,7) facing up. One static treasure cell is sampled from a finite
Gaussian centered at (4,4) with standard deviation 2.5. Ordinary
non-terminal actions sample a transient blue clue from the finite
Gaussian centered on the treasure with standard deviation 1.35. After
a failed PICKUP, the same clue distribution is conditioned on cells
visible in the current egocentric 3x3 view. The failure gives -1 and
play continues; success gives +1 and terminates. Only the treasure and
successive clue cells VARY between episodes.
```

```
Use the transient blue clues to locate and pick up the hidden treasure.
```

Figure 18: `TreasureHunt`: description and goal supplied to the model generators. Top: environment description. Bottom: goal.

F.4 Crafter Environments

The descriptions and goals of Figures 19 and 20 specify the two Crafter tasks; the description of `Collect Iron` calls the task Iron Hunt.

```
Crafter Zombie Escape is a 14 by 11 world. The player starts at (3,5),
faces right, has health 5, and publicly observes a world-aligned
X-major 9 by 7 crop plus health and wood. Actions are 0 wait, 1 left,
2 right, 3 up, 4 down, and 5 do. The terrain is a fixed asymmetric
grass meadow with paths and an impassable central rock formation.
Three trees are sampled without replacement on the eastern edge at
(13,y), y=2..8. One health-5 zombie starts on a walkable cell with x
in 6..9. It follows native stochastic Crafter pursuit: usually chase
within range, otherwise wander; it cannot enter blocked or occupied
cells, attacks an adjacent player for 2 health, and then has a
5-update attack cooldown. The player cannot damage it. Movement sets
an absolute facing direction and enters only grass/path/sand. Do
harvests the tree one cell ahead into grass and increments wood and
collect_wood. Food, drink, and energy never change. Player health
recovers by one after the native recovery counter passes 25, capped at
9.
```

```
Collect one wood while alive. A newly collected wood gives reward +1;
each point of damage gives -0.1. The episode ends on collect_wood or
death. Visible inventory keys are health and wood.
```

Figure 19: `Zombie Escape`: description and goal supplied to the model generators. Top: environment description. Bottom: goal.

```
Crafter Iron Hunt is a 14 by 11 world. The player starts at (1,5),
faces right, has health 9 and one hidden stone_pickaxe, and publicly
observes a world-aligned X-major 9 by 7 crop plus health, iron, and
wood. Actions are 0 wait, 1 left, 2 right, 3 up, 4 down, and 5 do. The
fixed base rows, from y=0 to 10, are: ttgggggggMMMM / ggttggggMMMMMM
/ tgtgggggMMMMMM / gtgggggssMMMM / gggtgggMMssMMM / gggggggpMMpppMM /
gggggpppMppMMM / ggggpsspppMMMM / wwgggpppppMMMM / wwwwpppMMpMMMM /
wwwwppppMMMppM. Here g is grass, p path, w water, t tree, and both M
and s are stone. Six coal and two iron cells are sampled without
replacement from the 51 M cells; all remaining M cells are stone. One
health-5 native zombie starts on a non-start grass cell. It uses
native stochastic pursuit and adjacent attacks for 2 health with
cooldown 5. The player may attack the zombie in front for one health
per do. Movement enters only grass/path/sand. Do on iron in front
consumes no pickaxe, turns it into path, increments iron and
collect_iron; the stone pickaxe is required and retained. Do on a tree
collects wood and turns it into grass. Life stats are fixed except
health and native health recovery.
```

```
Collect one iron while alive. A newly collected iron gives reward +1;
each point of damage gives -0.1. The episode ends on collect_iron or
death. Visible inventory keys are health, iron, and wood.
```

Figure 20: `Collect Iron`: description and goal supplied to the model generators. Top: environment description. Bottom: goal.

F.5 Offline Data

Each task has ten carefully collected offline episodes on seeds disjoint from evaluation. ABC and Pinductor learn from their observation histories, whereas POMDP-Coder receives the recorded native states and Tabular fits frequency tables for I, T, O, and R to them.

F.6 Implementation Details and Hyperparameters

Table 2: Settings on MiniGrid. Every model-based method plans with the best-first search of Appendix B.3 at an action cost of 0.01; the search budget counts node expansions per decision. The LLM methods use `gpt-5.6-luna` with reasoning effort `xhigh` through the Codex CLI.

	ABC	**Pinductor**	**POMDP-Coder**	**Tabular**
Particles K	10	10	10	10
Search budget	2,000	5,000	5,000	5,000
Predicate budget	64	–	–	–

ABC scores a candidate with $K = 10$ on the training and validation parts, each as the mean of Equation 7 over episodes with $\lambda = 0.01$ and the first term floored at $F_{\min} = -100$, then averages the two parts. If no drawn candidate explains all offline episodes, the draw uses all scored candidates.

Belief-update analysis. The analysis of Section 4.2 runs only the filters of ABC-GT and Pinductor-GT, with $K = 10$ and no observation at reset, on 100 histories per task. These are carefully collected trajectories that eventually reveal the complete state. Confidence bands are pointwise 95% percentile intervals from 5,000 bootstrap resamples of histories.

Particle-budget sweep and Crafter. The sweep of Section 4.3 runs ABC-GT and Pinductor-GT on seeds 0–29 with $K \in \{1, 5, 10, 20, 30\}$, a prior-proposal limit of $5{,}000K$, and the search budgets of Table 2. Crafter uses the same search budgets with an action limit of 100, $K = 10$, and a prior-proposal limit of 50,000; its models are generated with `gpt-5.6-sol` at reasoning effort `medium`. ABC generates one model per task with the MiniGrid budgets, and Pinductor fits one model per task offline, updates it at most once from an adaptation episode on seed 1010, and then freezes it.

G Detailed Experimental Results

G.1 Planning Performance on MiniGrid

Table 3: Success rate on MiniGrid (Figure 4). Success rate (%) over 30 episodes per environment.

Environment	ABC-GT	Pinductor-GT	ABC	Pinductor	POMDP-Coder	Tabular	Random
Unlock	**100.0**	**100.0**	**100.0**	90.0	**100.0**	10.0	0.0
MultiKeyUnlock	**93.3**	76.7	90.0	53.3	33.3	0.0	0.0
MovingBall	**100.0**	63.3	**100.0**	60.0	73.3	10.0	13.3
MultiMovingBalls	**93.3**	40.0	86.7	33.3	26.7	20.0	10.0
DoorMaze	**100.0**	46.7	43.3	40.0	0.0	0.0	0.0
TreasureHunt	**56.7**	3.3	33.3	0.0	20.0	3.3	3.3

Table 4: Episode length on MiniGrid (Figure 4). Mean episode length with 95% bootstrap confidence intervals over 30 episodes per environment.

Environment	ABC-GT	Pinductor-GT	ABC	Pinductor	POMDP-Coder	Tabular	Random
Unlock	**15.6** [14.4, 17.0]	16.0 [14.4, 17.8]	16.0 [14.5, 17.4]	24.8 [15.6, 41.9]	15.9 [14.7, 17.0]	95.0 [87.8, 100.0]	100.0 [100.0, 100.0]
MultiKeyUnlock	**22.3** [16.5, 29.1]	39.9 [28.3, 52.6]	33.4 [26.3, 40.1]	56.5 [39.1, 74.2]	72.4 [54.7, 89.6]	100.0 [100.0, 100.0]	100.0 [100.0, 100.0]
MovingBall	21.7 [17.5, 26.2]	29.5 [23.2, 36.0]	**21.1** [16.8, 26.1]	31.0 [26.4, 35.3]	33.7 [27.0, 40.0]	47.3 [42.8, 50.0]	46.9 [43.6, 50.0]
MultiMovingBalls	**20.8** [16.3, 25.6]	32.0 [25.3, 38.3]	22.4 [16.8, 27.9]	35.6 [30.3, 40.9]	39.3 [32.4, 45.4]	42.5 [38.1, 46.3]	44.6 [41.2, 47.6]
DoorMaze	**62.7** [53.1, 73.8]	126.9 [100.1, 153.4]	136.1 [87.4, 183.9]	138.5 [110.3, 164.3]	200.0 [200.0, 200.0]	200.0 [200.0, 200.0]	200.0 [200.0, 200.0]
TreasureHunt	**61.6** [49.0, 74.3]	98.5 [95.6, 100.0]	74.6 [58.5, 87.7]	100.0 [100.0, 100.0]	93.4 [87.5, 98.6]	98.0 [94.1, 100.0]	98.5 [95.4, 100.0]

G.2 BELIEF UPDATES

Tables 5 and 6 report Figures 5 and 6 at five points along each history. Progress 0 is the first update and progress 1 the last.

Table 5: Mass on True State (Figure 5). Mean probability assigned to the true complete state, with 95% bootstrap intervals in brackets.

		Progress				
Environment	**Method**	**0**	**0.25**	**0.5**	**0.75**	**1**
Unlock	ABC-GT	0.358 [0.283, 0.438]	0.948 [0.912, 0.978]	1.000 [1.000, 1.000]	1.000 [1.000, 1.000]	1.000 [1.000, 1.000]
	Pinductor-GT	0.163 [0.126, 0.203]	0.906 [0.863, 0.945]	1.000 [1.000, 1.000]	1.000 [1.000, 1.000]	1.000 [1.000, 1.000]
MultiKeyUnlock	ABC-GT	0.048 [0.025, 0.078]	0.851 [0.794, 0.903]	1.000 [1.000, 1.000]	1.000 [1.000, 1.000]	1.000 [1.000, 1.000]
	Pinductor-GT	0.007 [0.001, 0.017]	0.168 [0.102, 0.241]	0.200 [0.120, 0.280]	0.212 [0.133, 0.296]	0.212 [0.133, 0.296]
MovingBall	ABC-GT	0.240 [0.162, 0.318]	0.454 [0.372, 0.540]	0.614 [0.537, 0.694]	0.834 [0.775, 0.888]	1.000 [1.000, 1.000]
	Pinductor-GT	0.039 [0.023, 0.057]	0.115 [0.084, 0.149]	0.238 [0.186, 0.294]	0.310 [0.256, 0.364]	0.824 [0.756, 0.891]
MultiMovingBalls	ABC-GT	0.002 [0.001, 0.003]	0.065 [0.038, 0.098]	0.237 [0.174, 0.307]	0.534 [0.453, 0.615]	1.000 [1.000, 1.000]
	Pinductor-GT	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]	0.010 [0.000, 0.030]
DoorMaze	ABC-GT	<0.001 [<0.001, <0.001]	<0.001 [<0.001, <0.001]	0.019 [0.010, 0.032]	0.424 [0.366, 0.487]	1.000 [1.000, 1.000]
	Pinductor-GT	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]
TreasureHunt	ABC-GT	0.020 [0.017, 0.023]	0.572 [0.495, 0.649]	0.670 [0.580, 0.756]	0.684 [0.593, 0.772]	0.689 [0.599, 0.778]
	Pinductor-GT	0.027 [0.014, 0.043]	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]	0.000 [0.000, 0.000]

Table 6: Cumulative prior proposals (Figure 6). Mean cumulative number of candidate states drawn from I, including initialization, over 100 histories, with 95% bootstrap intervals in brackets. Figure 6 plots the reciprocal of these means. Pinductor-GT's limit is 50,000 per episode.

		Progress				
Environment	**Method**	**0**	**0.25**	**0.5**	**0.75**	**1**
Unlock	ABC-GT	269 [213, 326]	375 [318, 434]	375 [318, 434]	375 [318, 434]	375 [318, 434]
	Pinductor-GT	1,392 [1,167, 1,638]	8,261 [7,554, 8,958]	9,158 [8,490, 9,849]	9,158 [8,490, 9,849]	9,158 [8,490, 9,849]
MultiKeyUnlock	ABC-GT	320 [268, 374]	372 [319, 425]	372 [319, 425]	372 [319, 425]	372 [319, 425]
	Pinductor-GT	10,519 [8,077, 13,265]	49,608 [49,132, 49,952]	50,000 [50,000, 50,000]	50,000 [50,000, 50,000]	50,000 [50,000, 50,000]
MovingBall	ABC-GT	10 [10, 10]	10 [10, 10]	10 [10, 10]	10 [10, 10]	10 [10, 10]
	Pinductor-GT	26 [21, 31]	1,228 [351, 2,445]	6,835 [4,108, 9,934]	14,816 [11,030, 18,821]	23,680 [19,463, 27,968]
MultiMovingBalls	ABC-GT	10 [10, 10]	10 [10, 10]	10 [10, 10]	10 [10, 10]	10 [10, 10]
	Pinductor-GT	90 [61, 123]	35,937 [31,918, 39,701]	49,937 [49,810, 50,000]	50,000 [50,000, 50,000]	50,000 [50,000, 50,000]
DoorMaze	ABC-GT	10 [10, 10]	10 [10, 10]	10 [10, 10]	10 [10, 10]	10 [10, 10]
	Pinductor-GT	22 [20, 24]	28,907 [25,289, 32,447]	50,000 [50,000, 50,000]	50,000 [50,000, 50,000]	50,000 [50,000, 50,000]
TreasureHunt	ABC-GT	10 [10, 10]	10 [10, 10]	10 [10, 10]	10 [10, 10]	10 [10, 10]
	Pinductor-GT	15 [12, 18]	50,000 [50,000, 50,000]	50,000 [50,000, 50,000]	50,000 [50,000, 50,000]	50,000 [50,000, 50,000]

G.3 Particle Budget

Table 7: Success versus particle count (Figure 7). Success rate (%) over 30 episodes for each particle count K.

	ABC-GT					Pinductor-GT				
Environment	$K=1$	$K=5$	$K=10$	$K=20$	$K=30$	$K=1$	$K=5$	$K=10$	$K=20$	$K=30$
Unlock	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
MultiKeyUnlock	90.0	86.7	93.3	86.7	90.0	36.7	36.7	76.7	66.7	86.7
MovingBall	100.0	100.0	100.0	100.0	100.0	33.3	40.0	63.3	63.3	70.0
MultiMovingBalls	93.3	93.3	93.3	93.3	93.3	26.7	26.7	40.0	50.0	60.0
DoorMaze	100.0	100.0	100.0	100.0	100.0	50.0	76.7	46.7	70.0	60.0
TreasureHunt	6.7	46.7	56.7	66.7	76.7	3.3	0.0	3.3	3.3	0.0

G.4 Planning on Crafter

Table 8: Planning on Crafter (Figure 9). Success rate (%) over 30 episodes per task, with the number of successes in parentheses.

Task	ABC-GT	Pinductor-GT	ABC	Pinductor
Zombie Escape	**73.3** (22)	46.7 (14)	33.3 (10)	23.3 (7)
Collect Iron	**33.3** (10)	10.0 (3)	**33.3** (10)	13.3 (4)

H BELIEF VISUALIZATIONS

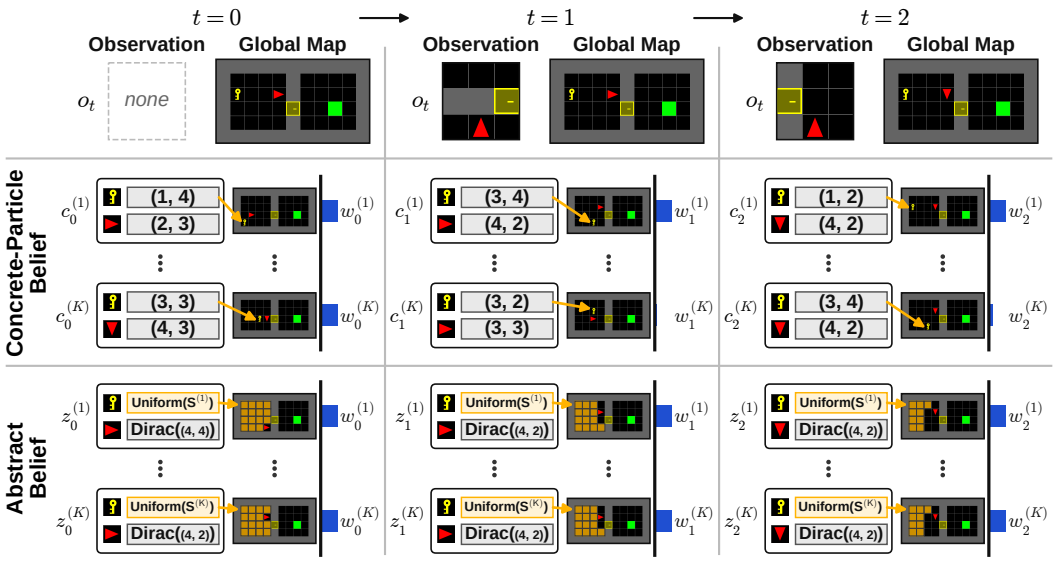


Figure 21: Concrete-particle belief (Pinductor) and abstract belief (ABC) on `Unlock`.

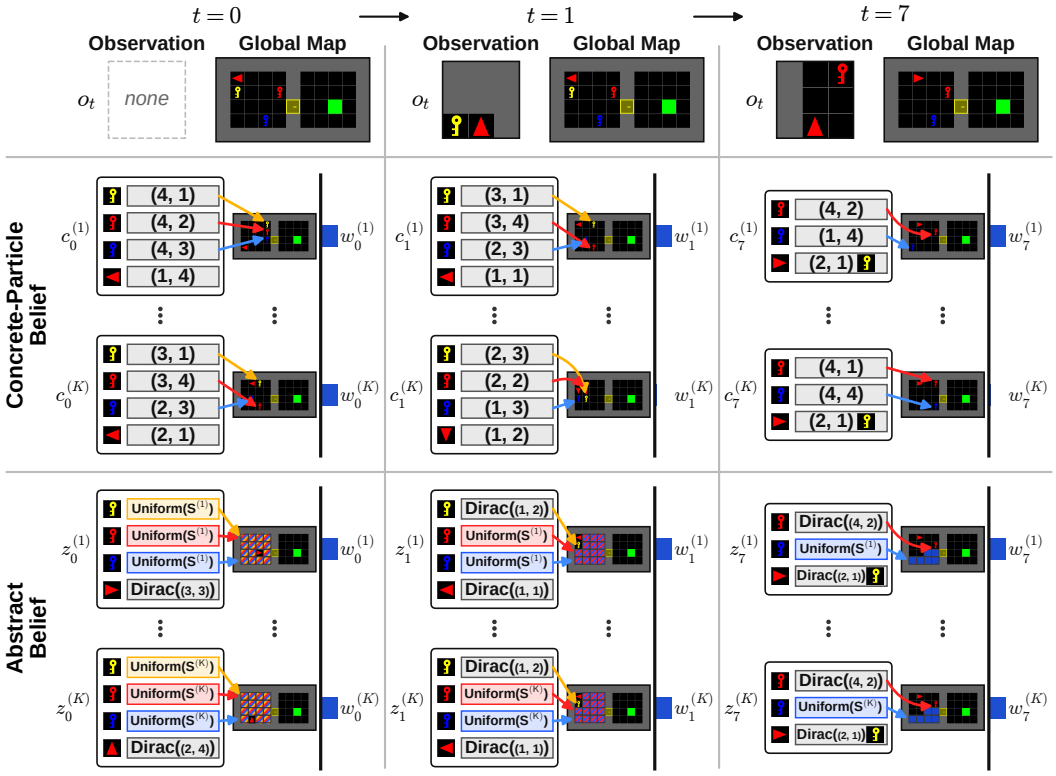


Figure 22: Concrete-particle belief (Pinductor) and abstract belief (ABC) on `MultiKeyUnlock`.

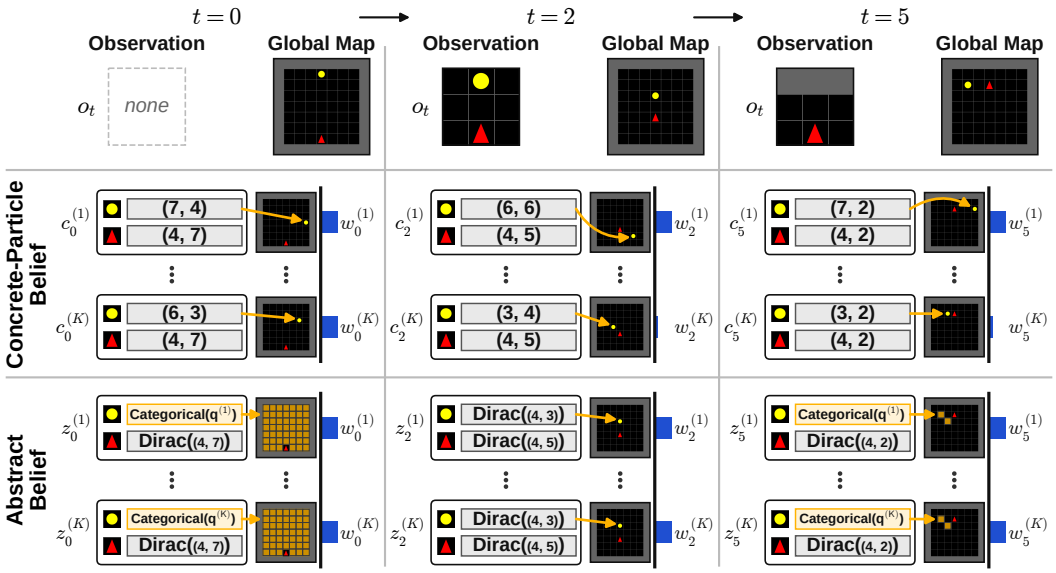


Figure 23: Concrete-particle belief (Pinductor) and abstract belief (ABC) on `MovingBall`.

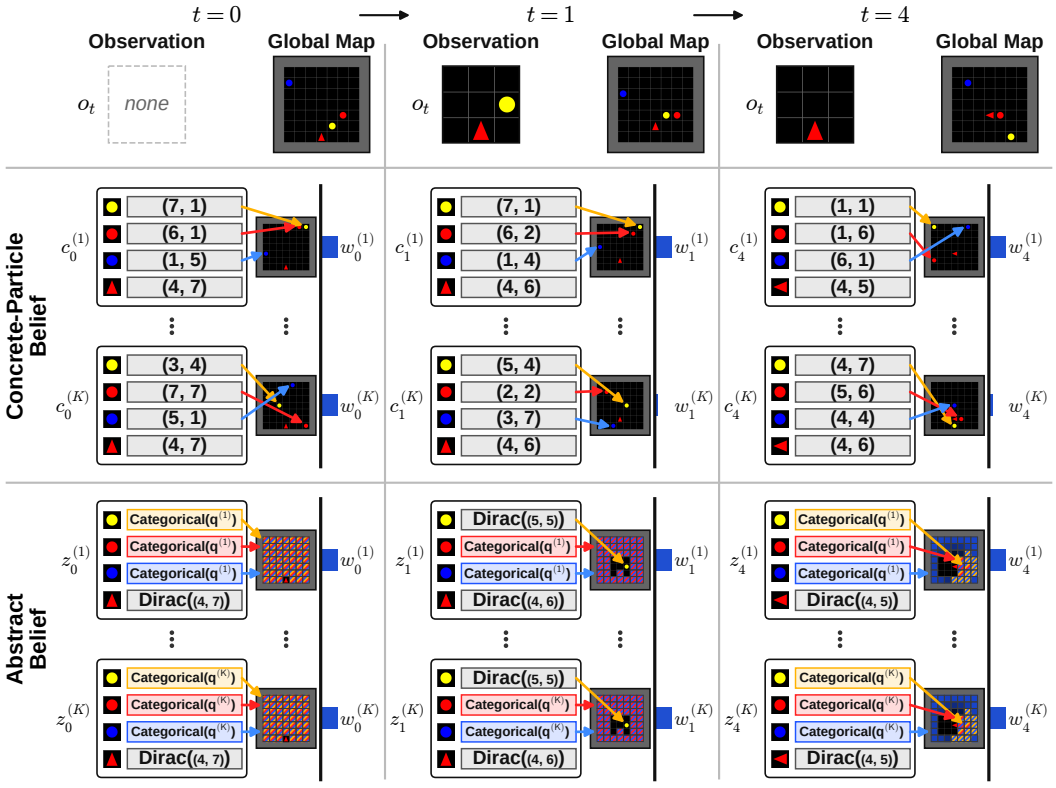


Figure 24: Concrete-particle belief (Pinductor) and abstract belief (ABC) on `MultiMovingBalls`.

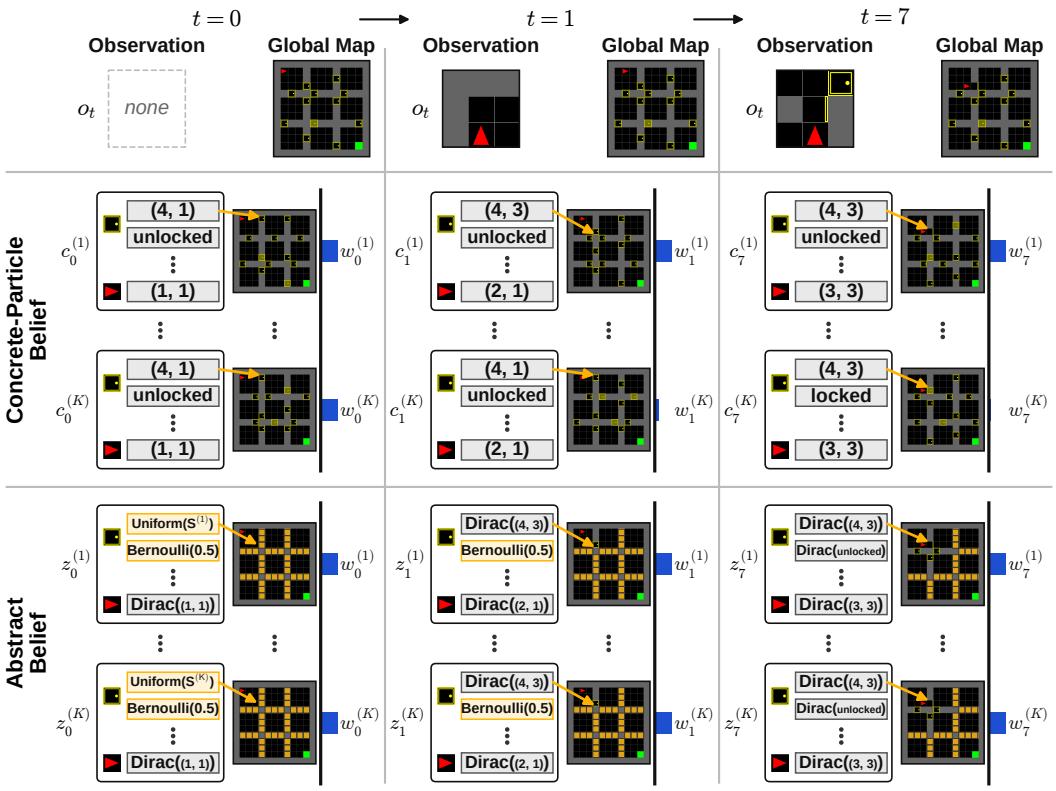


Figure 25: Concrete-particle belief (Pinductor) and abstract belief (ABC) on `DoorMaze`.

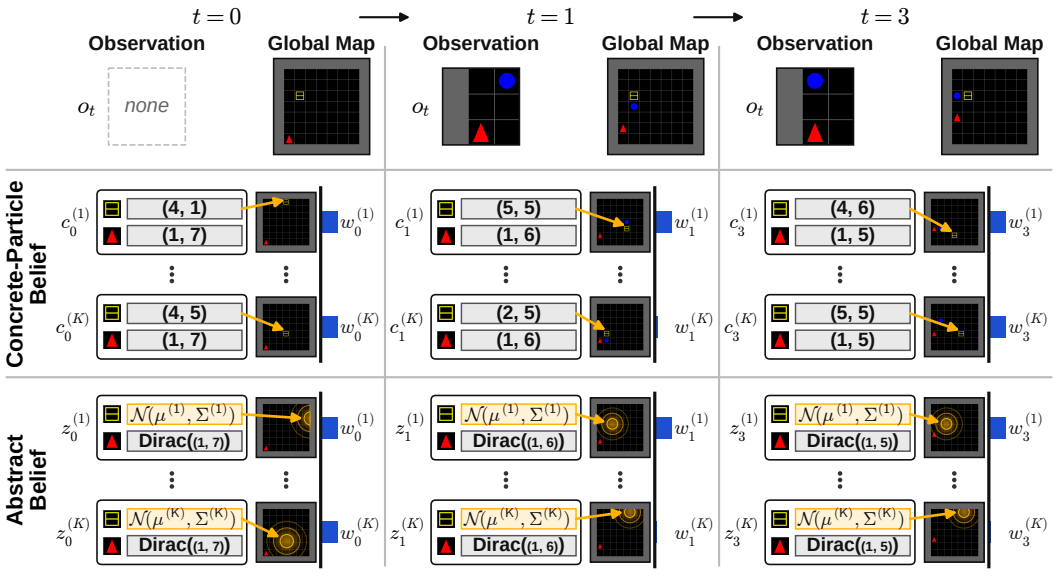


Figure 26: Concrete-particle belief (Pinductor) and abstract belief (ABC) on `TreasureHunt`.

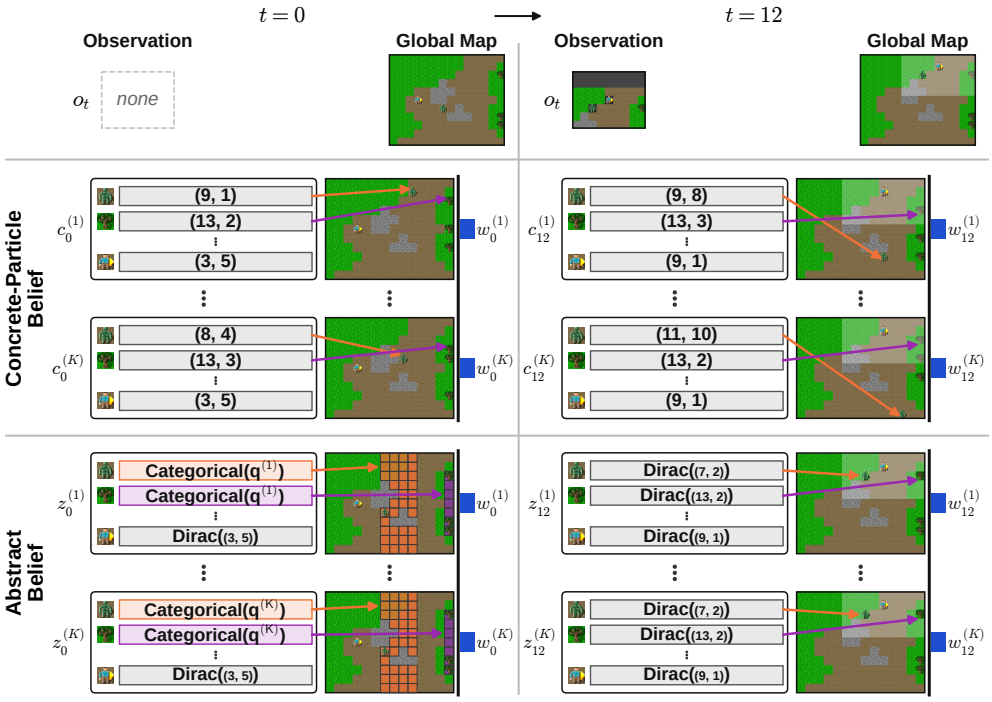


Figure 27: Concrete-particle belief (Pinductor) and abstract belief (ABC) on `Zombie Escape`.

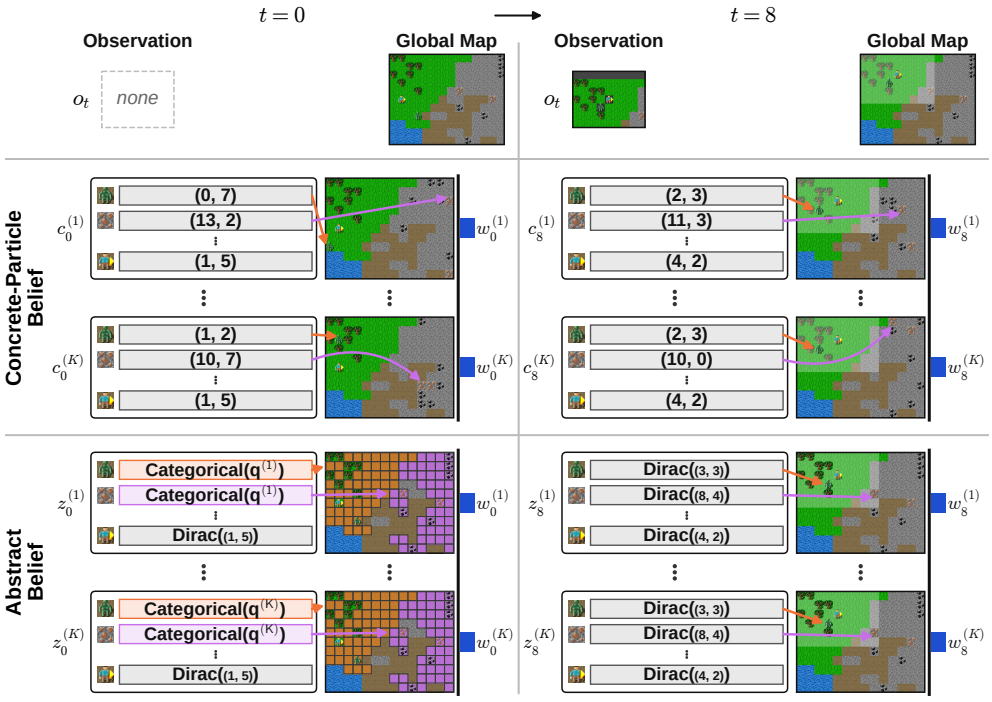


Figure 28: Concrete-particle belief (Pinductor) and abstract belief (ABC) on `Collect Iron`.