Algebraic World Model: From Exploration to Proofs

Guancheng Pan*
Zhejiang University
3250101086@zju.edu.cn

Qixin Tan*
Tsinghua University
tqx23@mails.tsinghua.edu.cn

Yang Yuan†
Tsinghua University
Shanghai QiZhi Institute
yuanyang@tsinghua.edu.cn

Andrew C. Yao†
Tsinghua University
Shanghai QiZhi Institute
andrewcyao@mail.tsinghua.edu.cn

Abstract

AI has made remarkable progress in mathematics, solving challenging problems with machine-checkable proofs in Lean, yet typically relying on predefined mathematical definitions and formal specifications. This raises a natural question: can it reason rigorously about an unknown world? We introduce the Algebraic World Model (AWM), a framework that turns interaction with unfamiliar environments into explicit mathematical models supporting formal reasoning. AWM explores the environment and develops executable Python hypotheses, continually testing and revising them against observations. It then structures these hypotheses into typed objects and shared rules, simplifies their structure through category-guided transformations, and formalizes the resulting model in Lean 4 to prove reachability and impossibility. AWM can formally prove that a goal is unreachable, even beyond explored trajectories. On two impossible game levels, it constructs closed, kernel-checked Lean proofs excluding every legal winning sequence in all six runs, while Claude Opus 5.5 and GPT-6 Astra, equipped with coding tools, produce none in twelve baseline runs. Four of AWM's six initial bounds are correct, compared with one for Opus 5.5 and none for GPT-6 Astra; given only feedback that a cheaper win exists, AWM revises its rules and constructs new proofs, reaching six of six correct bounds. Moreover, AWM produces interpretable mathematical models whose rules can be inspected and tested step by step. Across all 25 public ARC-AGI-3 games, these models enable Lean to replay the winning trajectories of all 183 levels and generalize beyond observed play, predicting unseen transitions and detecting changes in game mechanics. With Claude Opus 5, AWM clears all levels with a score of 99.9, while its certified winning trajectories use only 43% of the human baseline's actions.

1 Introduction

Artificial intelligence systems now produce Lean proofs of mathematical olympiad problems and some open Erdős problems (Achim et al., 2025; Hubert et al., 2026; Sothanaphan, 2026; Tsoukalas et al., 2026). These advances typically rely on predefined mathematical definitions and formal specifications. In an unfamiliar environment, however, an agent must first identify the relevant objects, infer how actions change them, and determine what constitutes success. The rules on which its reasoning depends are themselves hypotheses to be learned and revised through interaction. This raises a broader question:

Can an agent reason rigorously about an unknown world?

*These authors contributed equally.
†Corresponding authors.

Euler's treatment of the Königsberg bridges illustrates the role of mathematical structure (Euler, 1741). Land masses become vertices and bridges become edges. A walk crossing every bridge exactly once pairs arrivals and departures at every vertex other than its endpoints, so at most two vertices can have odd degree. The Königsberg graph has four, ruling out every such walk without enumerating routes. A winning trajectory likewise establishes reachability, but proving that none exists requires reasoning about all legal sequences. When the environment is unfamiliar, the objects and rules needed for that reasoning must themselves be learned.

We introduce the Algebraic World Model (AWM), a framework that learns explicit mathematical models of unfamiliar environments through interaction and uses them for formal reasoning. AWM proceeds through three stages (Figure 1). In **Explore & Prototype**, the agent actively probes the environment and maintains executable hypotheses, testing and revising them against observed transitions. **Structure & Simplify** organizes these hypotheses into typed objects and rules, then uses category-guided transformations, each checked in Lean through limits and colimits, to identify shared structure and simplify its representation while preserving tested distinctions. In **Formalize & Prove**, AWM completes the model in Lean 4 (de Moura & Ullrich, 2021) and constructs kernel-checked proofs of reachability, impossibility, and other properties. A winning witness establishes reachability; an invariant or a suitable cost lower bound can exclude every legal winning sequence under the model.

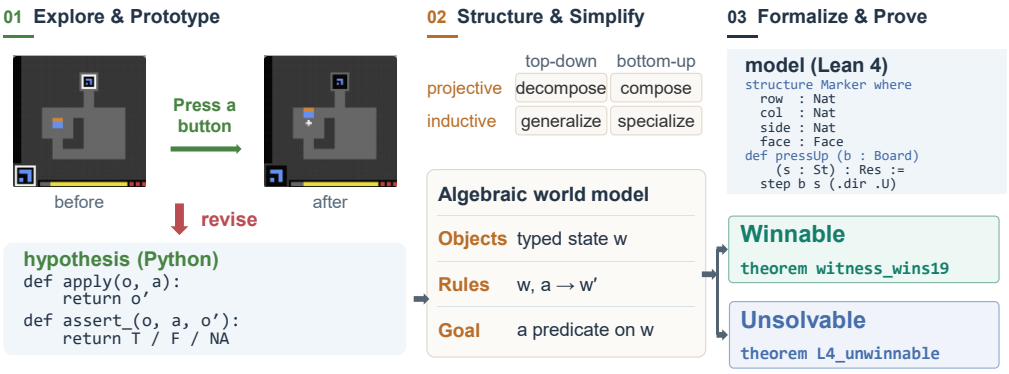

Figure 1: AWM's three-stage pipeline: interactive exploration builds a provisional executable model, algebraic transformations reveal and simplify its structure, and Lean 4 formalization enables certified proofs of reachability or impossibility.

We test AWM's ability to prove impossibility on ARC-AGI-3 (ARC Prize Foundation, 2026b), whose games reveal neither their rules nor their goals. We construct two impossible variants of `sc25` levels, preserving their mechanics while setting action budgets just below the minimum winning costs (§4). Across three runs per level, AWM produces closed, kernel-checked Lean proofs excluding all legal winning sequences under its learned models in all six runs. Claude Opus 5.5 and GPT-6 Astra produce no such proofs across twelve baseline runs.

The difference extends beyond proof construction. AWM's initial bounds match the minimum winning costs in four of six runs, compared with one for Opus 5.5 (without a proof) and none for Astra; other baseline runs report incorrect bounds or remain inconclusive. For example, on level 4 Astra claims impossibility with an unproved minimum winning cost of 22, whereas AWM proves a tight bound of 19 under its learned model, matching the game's actual minimum winning cost. Given only feedback that a cheaper win exists, AWM revises the rules underlying its remaining two bounds and constructs new closed proofs, bringing its bound accuracy to six of six.

AWM's learned mechanisms are also explicit and inspectable: typed objects and named rules make their predictions available for step-by-step checking. Across all 25 public ARC-AGI-3 games, AWM builds compact models with a median of six object types and ten rules. These models enable Lean to replay the winning trajectories of all 183 levels, checking represented state transitions and final goal conditions (§3). With Opus 5, AWM clears all 25 games with a score of 99.9, competitive with leading code-executing systems; its certified winning trajectories use 43% of the human baseline's actions.

We make three contributions.

- **A framework from exploration to algebraic structure.** AWM turns interaction with an unknown environment into executable hypotheses and organizes them into compact models through category-guided transformations checked in Lean, presenting object types as limits and colimits. Shared, parameterized rules express related mechanisms in a common form, allowing their properties to be stated and proved uniformly (§2).
- **Formal impossibility reasoning in unknown worlds.** AWM constructs closed, kernel-checked Lean proofs that rule out all legal winning sequences under its learned rules and specified constraints, including sequences never explored. This establishes model-relative impossibility rather than inferring it from unsuccessful play (§4).
- **Inspectable models and traceable proofs.** Typed objects and named rules make learned dynamics understandable and their predictions checkable against observations. Lean replay connects the formal representation to recorded behavior, while explicit proof dependencies identify the rules on which a conclusion rests, supporting targeted empirical examination and revision (§3; §4.3).

2 Method

AWM aims to turn interaction with an unknown environment into a mathematical model that supports rigorous reasoning about both reachability and impossibility. This requires more than collecting rules that explain observed transitions: the rules must form a coherent representation of the environment, and that representation must support reasoning about action sequences the agent has never explored.

We first describe the target model (§2.1), then present the three stages that construct and use it (Figure 1): Explore & Prototype develops executable hypotheses through interaction; Structure & Simplify organizes them into a compact algebraic model; and Formalize & Prove completes the model in Lean 4 and establishes machine-checked results.

2.1 The Algebraic World Model

An algebraic world model describes an environment through typed objects, rules, and a goal predicate, represented in executable Python and formal Lean 4 to support both empirical testing and mathematical reasoning.

Objects. Object types define the structure and validity of states, providing a common vocabulary for describing the environment. A parser maps observations, together with any required history or auxiliary information, to typed states w.

Rules. Rules describe how actions transform these states. Each Python rule specifies an applicability scope and a checker that reports agreement, contradiction, or non-applicability on observed transitions; effect rules also predict successor states or their components. Together, the rules define legal actions, their effects and costs $\mathrm{cost}(w, a)$, while the goal predicate identifies successful states.

Related mechanisms can share a parameterized operation $r(\theta)$, with θ determined by the state. This allows common properties to be expressed and proved uniformly across different mechanisms.

Formal representation. The same model is formalized in Lean 4, with objects and rules encoded as definitions and properties of their dynamics expressed as theorems. Its executable representation supports testing against observations, while its formal representation supports kernel-checked proofs over legal action sequences (§2.4).

2.2 Explore & Prototype

Setting. We consider deterministic, goal-directed environments with partial observations, modeled in the POMDP setting (Kaelbling et al., 1998). At step t, the agent observes o_t, takes an action a_t, and receives o_{t+1} without access to the underlying state or transition function. Its experience forms a corpus $\mathcal{D}_t$ of distinct transitions $x = (o, a, o')$.

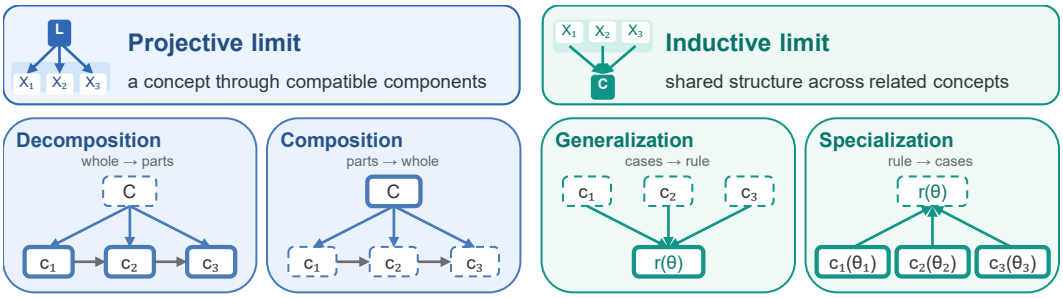

Figure 2: Four transformations guided by projective and inductive limits, applied to object types and rules. Dashed boxes mark removed components and bold boxes mark introduced components; arrows indicate relationships between components. Generalization uses $r(\theta)$, and an instance is written $r(\theta_i)$; specialization makes distinctions explicit through case-specific types or parameters.

Building and testing hypotheses. AWM actively explores the environment and maintains a provisional executable model as a *concept book*. Each concept is a Python hypothesis about an environmental mechanism, containing a natural-language claim, an applicability scope, supporting transitions, a checker, and an optional predictor.

For a concept c, its checker returns $v_c(x)$: `TRUE` if the transition agrees with the hypothesis, `FALSE` if it contradicts it, and `NA` if it falls outside its scope. The optional predictor specifies expected action effects for comparison with subsequent observations.

After each action batch, AWM rechecks the concept book $\mathcal{C}_t$ against the expanded corpus $\mathcal{D}_t$. Further interaction is allowed only when

$$\forall c \in \mathcal{C}_t,\ \forall x \in \mathcal{D}_t : \quad v_c(x) \notin \{\mathsf{FALSE}, \mathsf{ERR}\}, \tag{1}$$

where ERR denotes checker failure, including exceptions and timeouts.

This recheck ensures that contradicted or failing concepts must be revised or withdrawn before exploration continues. The agent then keeps revising its hypotheses and uses them to plan toward inferred goals or probe competing explanations.

The resulting prototype captures tested mechanisms but may describe related behaviors independently, motivating the next stage.

2.3 Structure & Simplify

Structure & Simplify turns the concept book's fragmented, overlapping descriptions into a compact model by identifying shared structure and reorganizing components through their relationships. AWM first extracts the required object types, constructs a parser from observations to typed states, and rewrites concepts as rules over these objects. It then restructures these *components*—object types and rules—through four *category-guided* transformations. A *projective limit* (limit) assembles compatible descriptions into a coherent whole, while an *inductive limit* (colimit) glues objects along shared structure (Mac Lane, 1998; Kashiwara & Schapira, 2006; Yuan, 2023). These two perspectives guide transformations based on compatibility and shared structure, respectively (Figure 2), and each transformation comes with an auxiliary Lean statement about cones, cocones and universal properties that the kernel checks (Appendix A.3).

Following projective limits, *decomposition* expresses a complex component through simpler components and their compatibility relations; Lean checks that it is the limit of the resulting diagram. Conversely, *composition* expresses a component through compatible existing components, moving the composite into the derived layer when a separate primitive definition is unnecessary; Lean checks that it is their limit.

Following inductive limits, *generalization* extracts shared structure from multiple components, expressing source rules c_i through a parameterized operation $r(\theta_i)$ with state-dependent θ_i; Lean checks that r is the copairing $[c_i]$ out of the coproduct of cases. Conversely, *specialization* refines overly

general components into more specific cases or introduces parameters to reduce exceptions and redundancy; Lean checks that the cases form a coproduct.

Preserving tested distinctions. Every transformation must compile its auxiliary Lean statement and preserve verdicts on recorded transitions. For rules, AWM also uses a diagnostic set $\mathcal{N}$ of perturbed recorded transitions to check whether simplification loses previously detected rejections.

Define $\mathrm{Rej}(c) = \{x \in \mathcal{N} : v_c(x) = \mathsf{FALSE}\}$ and $\mathrm{Rej}(S) = \bigcup_{c \in S} \mathrm{Rej}(c)$. Replacing rules S with T must preserve coverage of recorded transitions and satisfy

$$\mathrm{Rej}(T) \supseteq \mathrm{Rej}(S). \tag{2}$$

For object types, which have no checkers, transformations are evaluated by distinctions between recorded states, using state pairs as witnesses, and by rule verdicts once rules are available.

AWM applies these transformations and tests to object types, then rules, and finally rechecks object types against the settled rules.

2.4 Formalize & Prove

AWM completes the model in Lean 4 from the simplified objects and rules, encoding object types by their limit and colimit presentations, rules as `def` definitions, and derived properties as `theorem` statements; auxiliary proofs from Structure & Simplify stay in a separate layer. The transition function dispatches to named rules, while unresolved assumptions remain explicit theorem hypotheses rather than unchecked `axiom` declarations. Replaying recorded winning trajectories, the Lean model reproduces observed behavior on every component it represents, grounding its formal definitions in the dynamics the agent encountered. This enables theorems about all legal action sequences under the learned model, extending formal reasoning beyond the trajectories actually explored.

Proving reachability and impossibility. A level is *winnable* in the model if a legal action sequence reaches the goal. AWM proves reachability by constructing a witness checked in Lean. Impossibility can arise because the goal is unreachable, or because reaching it requires more cost than the available budget. AWM proves the former using an invariant that excludes the goal, and the latter using a lower bound on the cost of any winning sequence. For budgeted tasks, a potential V estimates the minimum remaining cost from each state and satisfies

$$V(w) \le \mathrm{cost}(w, a) + V(w') \quad \text{for every legal transition,} \qquad V(w) \le 0 \quad \text{on the goal,} \tag{3}$$

so every winning sequence costs at least $V(w_0)$, proving impossibility when this exceeds the budget. All proofs are closed and checked by the Lean kernel, without unproved placeholders (`sorry`) or kernel-bypassing computation (`native_decide`), establishing rigorous guarantees under the learned model's explicit premises.

Definition 1 (Optimal coverage)**.** A learned model covers the optimal winning sequences of a task if every such sequence in the environment is legal in the model, reproduces the environment's transitions step by step, and reaches the modeled goal. The task determines which winning sequences are optimal; whenever the task can be won, at least one is.

The model may omit any transition that no optimal winning sequence uses. Under optimal coverage, impossibility in the model transfers to the environment (Appendix A.4, Theorem A.1).

2.5 Example: one rule for four mechanisms

In game `lp85`, buttons move colored tiles, and a level is won when every slot holds its matching color. During Explore & Prototype, the agent discovers eleven concepts describing four button mechanisms: belts, stars, hubs, and panels (Figure 3a). In Structure & Simplify, AWM applies the generalization transformation to recognize that all four mechanisms permute tile positions, replacing the eleven concepts with a shared operation, `press_permutes`, parameterized by the pressed button. On the diagnostic set used for this merge, the shared rule retains all 55 rejections of the source concepts and rejects 21 additional corrupted transitions. The shared rule also shortens reasoning: because one color-count invariant covers every button, proofs that a variant is unsolvable need two to three times fewer lemmas than proofs over the eleven separate concepts (Appendix B.3). A separate check tests

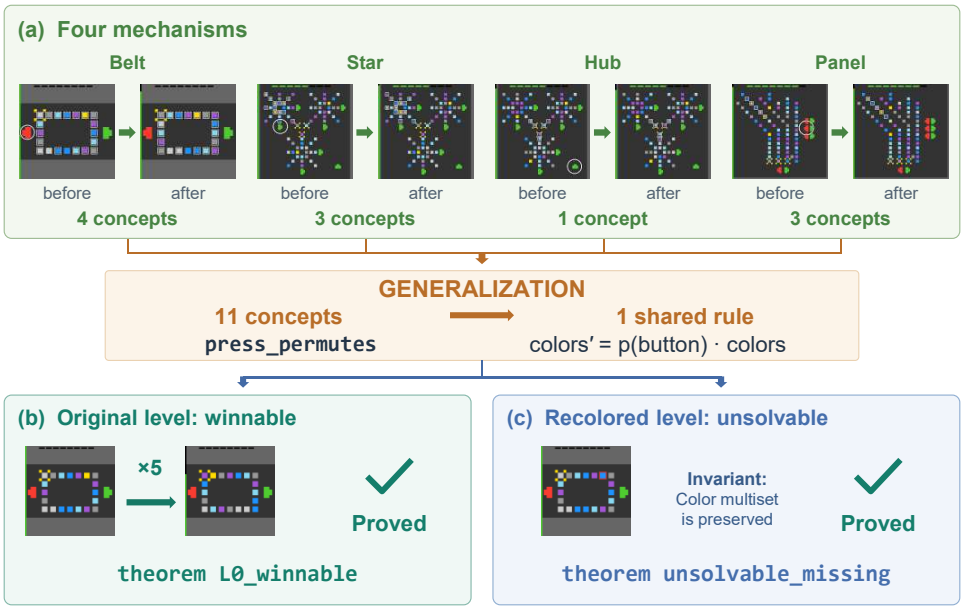

Figure 3: From shared structure to formal proofs in `lp85`. (a) Eleven concepts describing four button mechanisms are generalized into a shared permutation rule. Each before/after pair shows a recorded press, with the pressed button circled and changed tiles outlined. (b) A five-click sequence witnesses reachability. (c) Recoloring the only yellow tile makes the goal unreachable, since every press preserves color counts. Both conclusions are formally proved in Lean 4.

whether such a model-level proof also holds for the game: on the level-6 tile-swap variant U, Lean verifies its unsolvability proof against an engine-derived graph covering all 52,500 transitions among 7,500 configurations, with no axioms beyond Lean's standard ones (Appendix B.4).

In Formalize & Prove, Lean certifies a five-click winning sequence (Figure 3b) and proves a recolored variant unwinnable using the color-count invariant shared by all button permutations (Figure 3c). The final model contains six object types and six rules, and Lean replays all 199 recorded transitions without mismatch.

3 Twenty-five public games

ARC-AGI-3 (ARC Prize Foundation, 2026b) scores each level by the squared ratio of a human baseline's action count to the agent's, weights later levels more, and gives an uncleared level zero. We run AWM with Claude Opus 5 on all 25 public games (183 levels), starting each game with an empty concept book. AWM clears every level with a score of 99.9 and the fewest certified actions among the systems reporting this metric in Table 1. We further examine whether the resulting models are compact, reproduce observed behavior in Lean, and generalize beyond the trajectories from which they were learned.

Compact models that reproduce observed behavior. Across the 25 games, AWM records 742 executable concepts and withdraws 143 during model revision. The retained concepts are tested against the recorded transitions. Structure & Simplify organizes the rest into 170 object types and 283 rules, a median of 6 types and 10 rules per game. Shared patterns include translations, permutations and goal conditions. Lean replays the winning trajectory of every one of the 183 levels from its initial state and verifies its final state against the modeled goal (Table 2).

Generalization beyond observed transitions. We next test the learned rules on behavior the record never showed. We freeze the models of `lp85` and `ft09` after exploration and evaluate them on transitions from random play with previously unseen screens (seeds 0–9; Appendix B.2). The final

Table 1: Official scores on the 25 public ARC-AGI-3 games. Claude Opus 5 and GPT-6 Astra run under the official standard harness, which provides no code execution; the other systems execute code. AWM's certified clears follow uncertified sessions of the same run in 22 games. The actions for Rodionov's agent are from its ARC Prize scorecard and Memento 3's as reported in (Zhao et al., 2026).

System	Code	Mean score	Games at 100	Certified actions	Actions / human
Claude Opus 5	no	40.7	5	—	—
GPT-6 Astra	no	68.3	10	—	—
GPT-6 Astra (adapter) (ARC Prize Foundation, 2026a)	yes	100.0	25	—	—
Rodionov (2026b)	yes	99.0	23	8,347	0.49
Memento 3 (Zhao et al., 2026)	yes	100.0	25	7,518	0.44
AWM (ours)	yes	99.9	23	**7,448**	**0.43**

Table 2: Formalization and verification of the 25 learned world models in Lean 4. Axioms are those Lean's own logic uses; no proof adds axioms of its own.

Levels replayed	Rules formalized in Lean	Theorems	Axioms
183 / 183	220	1,374	only `propext`, `Classical.choice`, `Quot.sound`

frozen rules accept every valid transition in the new-trajectory sets. On the two modified `lp85` layout sets, they predict the next tile layout in all 4,160 cases; on the two changed-mechanic `ft09` sets, they reject all 4,031 transitions. On a third game, `ls20`, the same evaluation accepts 99.47–99.95% of valid transitions on previously unseen screens with the final rules, compared with 67–70% with the concepts before Structure & Simplify (Appendix B.2).

The changed mechanic modifies `ft09`'s hidden click pattern, causing each click to change 110 or 146 cells instead of 38; rejecting these transitions shows that the rules detect behavior inconsistent with the learned dynamics.

Clearing the games with efficient solutions. The same models are effective for play. AWM with Opus 5 achieves a score of 99.9, compared with 40.7 for Opus 5 under the official standard harness without code execution. It scores 100 on 23 games and at least 97.6 on every game, performing competitively with leading code-executing systems (Table 1). Its certified clears require 7,448 actions, only 43% of the human baseline's 17,135 and the fewest among systems reporting this metric, compared with 7,518 for Memento 3 (Zhao et al., 2026) and 8,347 for Rodionov (2026b). Including exploration and other sessions, AWM uses 24,604 engine actions in total (Appendix B.1). These results show that AWM can construct and use explicit world models while retaining competitive solution efficiency. §4 next examines whether such models also support formal proofs of impossibility.

4 Proving impossibility

A winning trajectory establishes reachability; impossibility requires excluding every legal winning sequence within the budget, including those never explored. We test whether AWM's learned algebraic models enable such exhaustive proofs, and whether those proofs hold for the game itself.

4.1 The game, the seal, and the baselines

In `sc25`, a block moves on a grid and casts spells through a 3×3 panel to reach the exit within the level's budget. Moves and panel clicks cost one unit, completing a spell one more, fuel refunds ten, and the winning move is free. We construct two variants of levels 4 and 5 by rearranging selected elements while preserving the game mechanics, then seal them at budgets of 18 and 35. Exhaustive search on the game engine (the underlying game program), covering all four initial facings and all 13 actions, establishes minimum winning costs of 19 and 36, respectively: neither sealed level can be won, while one additional unit permits a win.

Table 3: Impossibility reasoning across six runs per agent. Bound accuracy is evaluated against the games' minimum winning costs before and after feedback-guided repair. First claim averages the earliest impossibility claim among runs making one, including later withdrawals; it is not proof-completion time. Per-run results are in Appendix C.3.

Agent	Kernel-checked model proofs	Runs claiming impossible	Correct bounds initial → feedback	First claim (attempt)
Claude Opus 5.5	0/6	3/6	1/6 → 1/6	5.3
GPT-6 Astra	0/6	2/6	0/6 → 2/6	8.0
AWM	**6/6**	**6/6**	**4/6 → 6/6**	**2.0**

Claude Opus 5.5 and GPT-6 Astra receive the game interface and coding tools, including simulator construction and search; AWM uses Opus 5.5 with the procedure of §2 and a generic Lean skeleton. Each condition has three runs per level, with up to eight reset-delimited attempts per run, ending at game over or recorded abandonment. After each attempt, every agent is asked whether the sealed level can be won and to justify its verdict; AWM constructs Lean proofs under its learned model. Impossibility can be established by proving that the minimum winning cost exceeds the budget. We evaluate proof validity under the learned model separately from numerical agreement with the engine.

4.2 Certified lower bounds

Proving beyond explored trajectories. All six AWM runs produce closed, kernel-checked Lean proofs of impossibility under their learned models and explicit premises, while none of the twelve baseline runs produces such a proof (Table 3). Crucially, these proofs exclude all legal winning sequences within the budget under the learned models, including those never explored. For example, an AWM run for level 4 proves

$$\forall B \leq 18,\ \forall \mathbf{a}, \quad \mathrm{Won}(\mathrm{run}_B(w_0, \mathbf{a})) = \mathsf{false}, \tag{4}$$

and verifies a winning witness at $B = 19$. Together, the lower bound and witness certify a tight threshold under the model, whereas finding a winning route alone cannot exclude cheaper alternatives.

AWM establishes such bounds through potential or closure certificates over arbitrary legal action lists. Certificate data may come from any search; Lean checks that the certificate proves the stated bound under the formal model, so the proof need not trust the search implementation. In our mutation tests, the Lean verification package also detects reversed bounds and mismatches between the Python and Lean versions; a separate statement audit reports vacuous premises (Appendix C.7).

Beyond unproved numerical judgments. Only five of twelve baseline runs claim impossibility, four with incorrect bounds. Even the sole correct Claude Opus 5.5 bound lists "that my hand-counted path lengths are truly minimal" as *assumed, not checked*. Finding a winning route establishes an upper bound on the minimum cost, not its optimality. AWM instead provides kernel-checked lower-bound proofs under its learned models.

Figure 4 illustrates this distinction on L4. Astra finds a winning route costing 22 and assumes it is optimal without proof, incorrectly ruling out wins at budgets 19–21. In contrast, AWM formally proves a lower bound of 19 under its learned model and verifies a matching winning witness, establishing a tight bound that agrees with the game's minimum winning cost.

Proof validity and bound accuracy are distinct. Before feedback, four of six AWM bounds are correct, compared with one for Opus 5.5 and none for GPT-6 Astra.

Some runs identify a correct winning sequence but charge the otherwise free winning move, reporting a bound one unit higher. Because this cost cannot be observed on sealed levels, we count such bounds as correct for all agents only when the discrepancy is explicitly attributable to this accounting convention, not to an incorrect route or cost estimate.

The remaining two AWM bounds, 20 on L4 and 47 on L5, arise from overly restrictive learned rules that omit legal game transitions and violate model coverage (§2.4). Their proofs remain valid under those models, although their numerical thresholds disagree with the game.

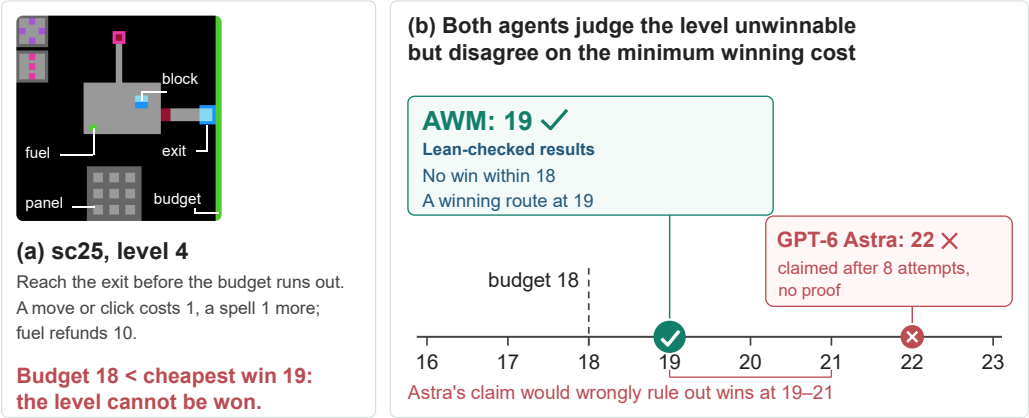

Figure 4: Sealed sc25 level 4. AWM proves in Lean that the minimum winning cost under its learned model is 19, one unit above the budget, and verifies a route at that cost; GPT-6 Astra claims 22 without a proof.

4.3 Auditing premises and proving again

After the initial evaluation, the two AWM runs with bounds of 20 and 47 receive feedback only that the game permits a cheaper win, without revealing its cost, route or the faulty rule. AWM revisits its explicit rules, tests alternative formulations through interaction, and constructs new proofs under the refined models (Appendix C.4).

On L4, AWM discovers that growing onto fuel collects it, reducing the bound from 20 to 19. On L5, it identifies terrain rather than limited range as the cause of a failed shot, reducing the bound from 47 to 37 under its convention of charging the winning move, equivalent to the game's minimum cost of 36. Both refinements succeed within four additional attempts, each yielding a new closed Lean proof. With this feedback, AWM reaches correct bounds in all six runs.

We provide the same discrepancy feedback and additional attempt budget to the four baseline runs with incorrect bounds. Astra corrects both bounds but produces no formal proofs, while Claude Opus 5.5 corrects neither.

The proof points to the error. Both repairs revise explicit rules used by the proofs. A separate injected-error study tests whether proof dependencies help target such checks. On L4, adding these dependencies to per-rule evidence yields a transition exposing the faulty rule in 29 of 42 runs, compared with 16 of 42 given per-rule evidence alone and 30 of 42 told the faulty rule. The dependency-guided condition raises no false alarms in 12 error-free controls (Appendix C.6).

4.4 From model proofs to the engine

Matching the game's minimum winning cost provides evidence of accurate reasoning, but does not establish that a formal proof under the learned model also holds for the game. To transfer such a proof, the learned model need not cover every game transition; by Definition 1, it suffices that it reproduce the game's optimal winning sequences. For the budgeted tasks here, we verify sufficient certificate-level conditions on engine-derived graphs, which preserve budget feasibility and the proof invariants (Lemmas A.2–A.3).

We exhaustively check the engines' budget-feasible transitions from the fixed opening against the potential inequalities and invariant conditions used in the model proofs. These checks establish that five of the six model-level impossibility proofs can be extended to the original games. Appendix C.5 gives the per-run transfer arguments, including the completed rules and certificates.

5 Related work

Executable world models. Code world models are learned from interaction for planning (Tang et al., 2024), refined by generation, testing and repair (Dainese et al., 2024) or composed from programmatic experts (Piriyakulkij et al., 2025); theory-based agents explore and plan with learned theories (Tsividis et al., 2021), and TheoryCoder-2 learns reusable abstractions for hierarchical planning (Ahmed et al., 2026). Such models are central to recent ARC-AGI-3 agents (Rodionov, 2026a; Courtis et al., 2026; Rodionov, 2026b): Twin does not act until its simulator reproduces every observed transition (Skoutnev et al., 2026), and Tycho finds that a closer match does not by itself yield better play (Lehmann et al., 2026). The closest work, Memento 3, compiles a natural-language rulebook into code and accepts an update only if an LLM judges it faithful and replay is exact (Zhao et al., 2026). Yet a code world model can pass a sampled-transition check at 100% accuracy and still lose because it misses a key rule (Aguilar Martín, 2026). AWM instead abstracts its rules into a shared structure, tests them beyond the record, and proves reachability and impossibility, checking the proofs' premises against the game.

Certified impossibility. Unsolvability certificates let an independent checker confirm that a planning task has no plan (Eriksson et al., 2017), in the spirit of certifying algorithms (McConnell et al., 2011), and certificate checking has been formally verified for temporal planning (Wang & Abdulaziz, 2025); all start from a supplied model. Impossibility transfers from an abstraction that covers the environment (Definition 1), with preserved initialization, goals and budget feasibility (Clarke et al., 1994); action-model learning gives conditions under which learned models support such transfer (Aineto & Scala, 2024). AWM learns its model from interaction; Lean checks every deduction from that model, and interaction checks the model's rules against the game.

Formal reasoning. Neural provers produce Lean proofs at olympiad level (Achim et al., 2025; Chen et al., 2025; Hubert et al., 2026; Ren et al., 2025), AI systems have produced Lean proofs of open Erdős problems (Sothanaphan, 2026; Tsoukalas et al., 2026), and autoformalization translates informal statements into formal ones (Wu et al., 2022). There the objects of proof are given mathematics; AWM must first infer the dynamics and costs its proofs quantify over, which makes the fidelity of its formal model an empirical question. Large-scale AI-assisted software verification proves properties of compiled programs relative to machine semantics and theorem statements, and can expose bugs during proof construction (Lucas et al., 2026). That work also studies Claude-generated x86 semantics; AWM learns the objects, action rules and goals of an unfamiliar interactive environment.

6 Limitations and conclusion

Limitations. We demonstrate AWM on ARC-like environments with discrete actions and immediate, observable feedback, where learned rules can be checked against recorded transitions; richer hidden state, delayed feedback and stochastic dynamics remain to be tested. Lean guarantees the correctness of proofs under the learned model's explicit premises, and whether these premises capture the environment is tested by interaction: the explicit rules and kernel-checked proofs separate errors in the learned model from errors in reasoning, so a discrepancy can be traced to its assumption and addressed by further exploration. Broader environments and tasks are the natural next test of such formal reasoning.

Conclusion. Formal reasoning traditionally begins with predefined objects and rules. AWM takes a step toward reasoning rigorously about unknown worlds by discovering these structures through interaction, organizing them into compact algebraic models, and formalizing them in Lean. The resulting models make learned mechanisms explicit and their behavior checkable step by step, while supporting kernel-checked proofs of reachability and impossibility, including claims about action sequences never explored. By making the assumptions behind these proofs explicit, AWM also allows discrepancies with the environment to be traced to learned rules and addressed through further exploration. Together, these capabilities bridge empirical exploration and formal reasoning, extending mathematical proof beyond problems whose definitions are given in advance.

Reproducibility statement

Public-game models and replays are available at https://github.com/Ephemeral6/awm-site. The readings of §2 and §3 come from the benchmark repository at commit `fa928185` (built 2026-10-24). Each game's model, tests and Lean project are regenerated by one command per game; the per-game Lean projects are checked with Lean 4.34.0 and do not use Mathlib. The readings of §4 come from the archive of the budget-sealed instance: both engine versions with their md5, the abstract model and its cross-check scripts with their raw outputs, the Lean files with their compile logs and axiom audits, every file each agent wrote, and the verbatim end-of-attempt interviews; the per-run readings are given in the appendix. The game environments are third party; official replays are linked rather than redistributed.

Statement on AI use

The agents studied in this paper are LLM agents, so AI use is part of the method. Generative AI was also used to draft and edit the text and to write the analysis scripts. All reported numbers come from script outputs over the recorded runs, not from model-generated text. The Lean proofs are checked by the Lean kernel, independently of any model.

References

Tudor Achim et al. Aristotle: IMO-level automated theorem proving. *arXiv preprint arXiv:2510.01346*, 2025.

Javier Aguilar Martín. When a verified world model still loses: Play-adequacy vs prediction-accuracy in LLM-synthesized code world models. *arXiv preprint arXiv:2607.14169*, 2026.

Zergham Ahmed, Kazuki Irie, Joshua B. Tenenbaum, Christopher J. Bates, and Samuel J. Gershman. Learning abstractions for hierarchical planning in program-synthesis agents. *arXiv preprint arXiv:2602.00929*, 2026.

Diego Aineto and Enrico Scala. Action model learning with guarantees. In *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 801–811, 2024. doi: 10.24963/kr.2024/75.

ARC Prize Foundation. GPT-6 Astra: ARC-AGI results. https://arcprize.org/results/openai-gpt-6-astra, 2026a. Public Demo, 25 environments; accessed 2 October 2026.

ARC Prize Foundation. ARC-AGI-3: A new challenge for frontier agentic intelligence. *arXiv preprint arXiv:2603.24621*, 2026b.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, et al. Seed-Prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint arXiv:2507.23726*, 2025.

Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994. doi: 10.1145/186025.186051.

David Courtis, Wenhao Li, and Scott Sanner. OPINE-World: Programmatic world modeling with ontology-error-prioritized interactive exploration for ARC-AGI-3. *arXiv preprint arXiv:2607.01531*, 2026.

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 37 (NeurIPS)*, volume 37, pp. 60429–60474, 2024. doi: 10.52202/079017-1933.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pp. 625–635. Springer, 2021.

Salomé Eriksson, Gabriele Röger, and Malte Helmert. Unsolvability certificates for classical planning. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 88–97, 2017. doi: 10.1609/icaps.v27i1.13818.

Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, 8:128–140, 1741. Volume dated 1736; Eneström index E53.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2026. doi: 10.1038/s41586-025-09833-y.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998. doi: 10.1016/S0004-3702(98)00023-X.

Masaki Kashiwara and Pierre Schapira. *Categories and Sheaves*, volume 332 of *Grundlehren der mathematischen Wissenschaften*. Springer, Berlin, Heidelberg, 2006.

Jens Lehmann, Andrei Aioanei, and Sahar Vahdati. Tycho: Active abstraction with programmatic world models for ARC-AGI-3. *arXiv preprint arXiv:2607.28287*, 2026.

Keane Lucas, Aengus Lynch, Luna Tong, Luke Champine, Zygi Straznickas, Dino Distefano, Harshikaa Agrawal, Evyatar Ben Asher, Ava Howell, Sami Jawhar, Chris Rohlf, Nickolai Zeldovich, Nicholas Carlini, Rajashree Agrawal, and Jason Gross. LLMs are making formal software verification feasible at a much larger scale. Anthropic research report, October 2026. URL https://www.anthropic.com/research/formal-verification. October 22, 2026.

Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, 2nd edition, 1998.

Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011. doi: 10.1016/j.cosrev.2010.09.009.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Wasu Top Piriyakulkij, Yichao Liang, Hao Tang, Adrian Weller, Marta Kryven, and Kevin Ellis. PoE-World: Compositional world modeling with products of programmatic experts. In *Advances in Neural Information Processing Systems 38 (NeurIPS)*, volume 38, 2025. doi: 10.52202/085713-0896.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

Sergey Rodionov. Executable world models for ARC-AGI-3 in the era of coding agents. In *Artificial General Intelligence: 19th International Conference, AGI 2026*, Lecture Notes in Computer Science. Springer, 2026a. arXiv:2605.05138.

Sergey Rodionov. Do coding agents need executable world models, simplification, and verification to solve ARC-AGI-3? *arXiv preprint arXiv:2607.15439*, 2026b.

Alexy Skoutnev, Kirill Acharya, Gaston Longhitano, Madeleine Udell, Kevin Ellis, and Iddo Drori. Twin: Playing an unknown game with a test-time digital twin. *arXiv preprint arXiv:2608.14490*, 2026.

Nat Sothanaphan. Resolution of Erdős problem #728: a writeup of Aristotle's Lean proof. *arXiv preprint arXiv:2601.07421*, 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 37 (NeurIPS)*, volume 37, pp. 70148–70212, 2024. doi: 10.52202/079017-2243.

Pedro A. Tsividis, Joao Loula, Jake Burga, Nathan Foss, Andres Campero, Thomas Pouncy, Samuel J. Gershman, and Joshua B. Tenenbaum. Human-level reinforcement learning through theory-based modeling, exploration, and planning. *arXiv preprint arXiv:2107.12544*, 2021.

George Tsoukalas et al. Advancing mathematics research with AI-driven formal proof search. *arXiv preprint arXiv:2605.22763*, 2026.

David Wang and Mohammad Abdulaziz. Formally verified certification of unsolvability of temporal planning problems. *arXiv preprint arXiv:2510.10189*, 2025.

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Advances in Neural Information Processing Systems 35 (NeurIPS)*, volume 35, 2022.

Yang Yuan. Succinct representations for concepts. *arXiv preprint arXiv:2303.00446*, 2023.

Haoyu Zhao, Zhengxu Yu, Zhiyuan He, Meng Fang, Rasul Tutunov, Haitham Bou-Ammar, Weilin Luo, and Jun Wang. Memento 3: Model-based recursive self-improvement through reflective rulebooks. *arXiv preprint arXiv:2610.11794*, 2026.

A METHOD DETAILS

This appendix gives the definitions behind §2: concepts and the gate (§2.2), the simplification test (§2.3), and the Lean statements, certificates and their transfer to the environment (§2.4).

A.1 Environment, concepts and the gate. We model an environment as a deterministic, goal-directed special case of a partially observable Markov decision process (Kaelbling et al., 1998): states $\mathcal{S}$, actions $\mathcal{A}$, observations $\mathcal{O}$, a transition function $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, an observation function $\rho : \mathcal{S} \to \mathcal{O}$ and goal states $G \subseteq \mathcal{S}$. The agent sees $o_t = \rho(s_t)$ and receives a signal when a level is cleared. A transition is a triple $x = (o, a, o')$, and a game's transition corpus $\mathcal{D}$ is the set of distinct transitions across its runs, deduplicated by content hash.

A concept $c = (\sigma_c, \alpha_c, \gamma_c, E_c)$ has an optional executable scope σ_c, an assertion α_c returning TRUE, FALSE or NA, an optional predictor γ_c, and a non-empty evidence set E_c. When the scope rejects x the verdict $v_c(x)$ is NA, otherwise it is $\alpha_c(x)$; an exception, an invalid return value or a timeout gives ERR. A concept is scored only on transitions outside E_c. The gate of Eq. (1) is enforced on the action command: it refuses the next action while the latest settlement has a FALSE or ERR verdict on any $x \in \mathcal{D}$, including the concept's own evidence. Figure 5 shows one concept.

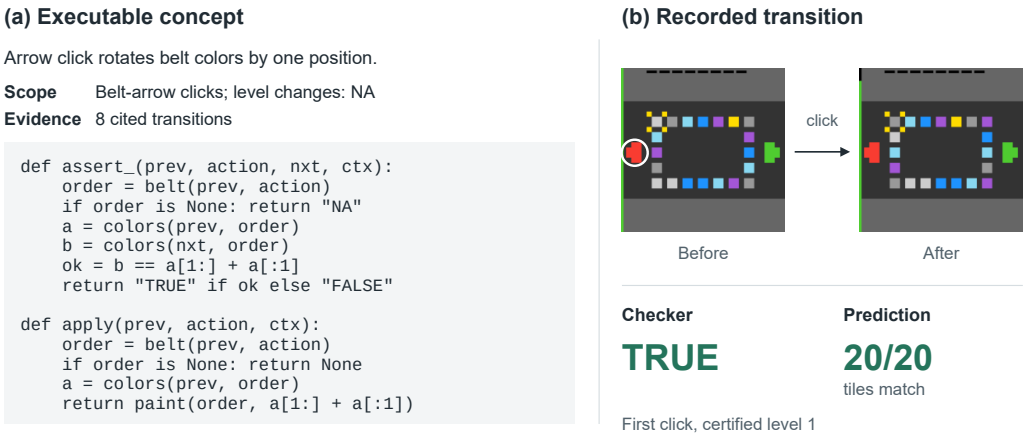

Figure 5: An executable concept in `lp85`. (a) Scope, evidence and abridged code for a click on an arrow pointing against its belt's order; helper names are simplified. (b) On the first click of the certified level-1 run, the checker returns `TRUE` and the prediction matches all 20 tiles.

A.2 Objects, rules and the simplification test. A partial parser $\pi : \widetilde{\mathcal{O}} \rightharpoonup \mathcal{W}$ maps an observation, augmented by any explicitly supplied history or auxiliary information, to a world of typed objects; $\pi(o)$ abbreviates parsing with that context. Pixel interpretation is confined to the parser, and parse failures are reported rather than guessed. A rule $r = (\sigma_r, \alpha_r)$ has a scope and an assertion on $(\pi(o), a, \pi(o'))$. The diagnostic set $\mathcal{N}$ of Eq. (2) contains at most 120 corrupted transitions per game, with a fixed seed. Each replaces part of a recorded transition (o, a, o') in one of five ways: the successor is o itself; the successor of the same action on another transition; the predecessor of another transition; o' with a few cells changed that do vary in the corpus; or another transition's o. We set

$$\mathrm{Rej}(r) = \{m \in \mathcal{N} : v_r(m) = \mathsf{FALSE}\}. \tag{A.1}$$

The following is an auxiliary diagnostic within the four transformations, not a separate operation or a sufficient deletion criterion: rule r has no unique rejection on $\mathcal{N}$ when

$$\mathrm{Rej}(r) \subseteq \textstyle\bigcup_{r' \neq r} \mathrm{Rej}(r'). \tag{A.2}$$

Any removal must preserve model behavior and pass the same before/after checks; an analogous field diagnostic fixes a field to a constant and compares verdicts on $\mathcal{D} \cup \mathcal{N}$. Figure 6 shows the test on `lp85`.

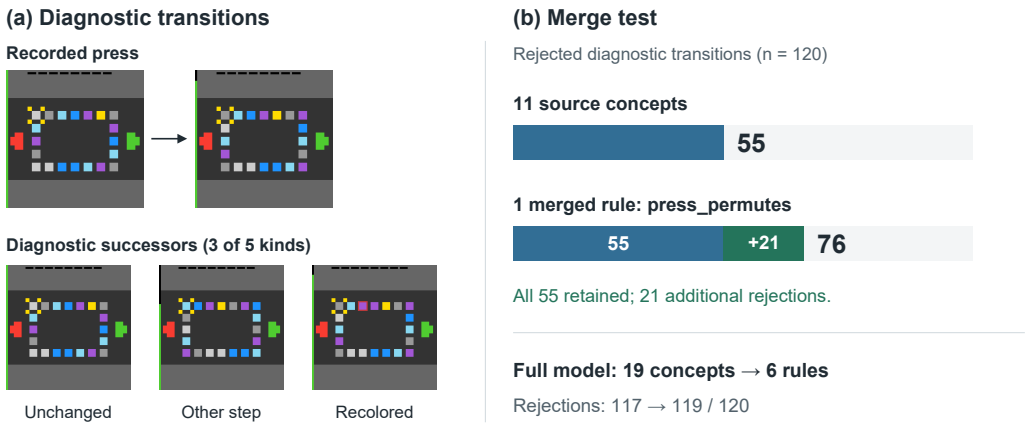

Figure 6: Testing a merge on `lp85`. (a) A recorded press and three of the five diagnostic constructions. (b) On the same 120 diagnostic transitions, `press_permutes` preserves all 55 rejections of its eleven source concepts and adds 21. Lack of unique rejections alone does not justify deletion: `dead_click`, `board_static` and `bar_never_drops` are retained based on further tests.

A.3 Presentations, transformations and their certificates. This paragraph states the definitions behind §2.3. Throughout, $\mathbf{Set}$ is the category of sets and functions; in Lean, objects are types and morphisms are functions.

Definition 2 (Diagram, cone, limit)**.** A diagram is a functor $D\colon J \to \mathbf{Set}$ from a small category J. A cone over D is a set L with maps $\pi_j\colon L \to D(j)$, $j \in J$, such that $D(f) \circ \pi_i = \pi_j$ for every $f\colon i \to j$. It is a limit if for every cone (L', π') there is a unique $u\colon L' \to L$ with $\pi_j \circ u = \pi'_j$ for all j. Dually, a cocone consists of maps $\iota_j\colon D(j) \to C$ with $\iota_j \circ D(f) = \iota_i$, and a colimit is a cocone through which every cocone factors uniquely.

In $\mathbf{Set}$, $\lim D$ is the set of compatible families $\{(x_j) \in \prod_j D(j) : D(f)(x_i) = x_j \ \forall f\colon i \to j\}$, and $\operatorname{colim} D$ is $\coprod_j D(j)$ modulo the equivalence relation generated by $x \sim D(f)(x)$. We use three index shapes: discrete (products and coproducts), the cospan $\bullet \to \bullet \leftarrow \bullet$ (pullbacks), and the parallel pair $\bullet \rightrightarrows \bullet$ (equalizers and coequalizers). A subtype $\{x \in X : \chi(x) = \mathsf{true}\}$ is the equalizer of χ and the constant map true, and a quotient X/R is the coequalizer of the two projections $R \rightrightarrows X$.

Definition 3 (Presentation)**.** A presentation of an object type T is a triple (D_T, C_T, u_T) of a diagram D_T whose values are previously introduced types or base types, a carrier C_T with a cone (or cocone) structure over D_T, and a proof u_T that this cone is a limit (or this cocone a colimit). The carrier is the type used by the computational layer: the universal property is proved for that type itself, not for a separate copy.

Two presentations of the same diagram have isomorphic carriers through a unique isomorphism compatible with the cones. For a quotient this yields the correspondence with a normal form: if the normal form, for instance a sorted list of colors, is also shown to be a colimit of the same parallel pair, the isomorphism with X/R is obtained from the two universal properties rather than constructed by hand. A map out of X/R is defined by lifting a map out of X, which requires a proof that it is constant on R.

Definition 4 (Case table)**.** A case table for a rule group with scope C is a finite set I with a classifier $\kappa\colon C \to I$. It satisfies the coproduct law on a corpus if every transition in C is classified (no gap), each transition receives one case (no overlap), and every case occurs; then $C = \coprod_{i \in I} \kappa^{-1}(i)$. A rule defined case by case is the copairing $[r_i]_{i \in I}$, the unique map whose restriction to each case is r_i.

Definition 5 (Category-guided transformation)**.** A transformation replaces components S of the model by components T, together with a representation map e from old to new states. It is one of the following.

- *Decomposition*: a component X is replaced by a diagram D and a cone (X, π) over D that is a limit.

- *Composition*: a component Y is defined as the limit of existing components under constraints $\varphi_1, \ldots, \varphi_k$, that is, the intersection of the subobjects $\{\varphi_i\}$; the composition is proper if no single φ_i cuts out Y alone (for $k = 2$, as in all compositions reported here, this is the same as each constraint being needed).
- *Generalization*: rules $r_1, \ldots, r_n$ on case sets $C_1, \ldots, C_n$ are replaced by the copairing $[r_i]$ on $\coprod_i C_i$.
- *Specialization*: a component X is replaced by a coproduct $\coprod_i X_i$ with a merge map $m\colon \coprod_i X_i \to X$, or by a family $X(\theta)$ with an added parameter.

It is accepted if (i) on the corpus $\mathcal{D}$ and the diagnostic set $\mathcal{N}$ the verdicts on recorded transitions are preserved, Eq. (2) holds, the task contracts hold under e (one step commutes with e; goal verdicts, checker verdicts and action costs are unchanged, so certified trajectories replay with their recorded action counts), and the kind-specific check listed below passes; and (ii) the corresponding Lean statement compiles at the same step. Deletion of a component that is a function of others, and passage to a quotient, are recorded separately and are not instances of the four transformations.

The kind-specific checks are reported by name, and a transformation that does not report all checks of its kind is rejected: decomposition reports a round trip; composition reports the equivalence $Y \Leftrightarrow \bigwedge_i \varphi_i$ and, for each constraint, a transition it admits and Y rejects; generalization reports per-case agreement and the coproduct law; specialization reports that merging the cases back reproduces the original verdicts and that collapsing the new parameter to a constant produces errors. Constraints that random perturbations rarely violate are tested on targeted perturbations that violate exactly one constraint.

Evidence. Each transformation records the kind of its evidence: *observed* (verdicts on the corpus), *finite exhaustive* (checked on every recorded board or state), or *for all well-formed states* (a Lean theorem). The last kind is admitted only if the cited theorem compiled at the step of the transformation and at least one cited theorem does not hold by definitional unfolding alone; for example, the equation stating that a rule defined by pattern matching equals the copairing of its branches holds by unfolding and certifies the form of the rule, not its content. Corpus evidence and Lean evidence answer different questions—whether the model agrees with the environment, and what the model entails—and neither substitutes for the other.

Dynamics. For a deterministic step $\delta\colon \mathcal{W} \times \mathcal{A} \to \mathcal{W}$, the free monoid $\mathcal{A}^*$ acts on $\mathcal{W}$ by $\varepsilon \cdot w = w$ and $(\bar{a}\, a) \cdot w = \delta(\bar{a} \cdot w, a)$; with partial legality, δ is a morphism of the Kleisli category of the option monad and the action is by partial maps (Moggi, 1991). A statement in this layer belongs to the model only if it is checked on every level of the corpus. In `lp85`, the rule is that a word of clicks acts by the composite of the clicked buttons' permutations, with clicks outside every button acting as the identity; Lean proves it for every board, and the corpus confirms it on all multi-step prefixes of every level. That the action on the first level factors through $\mathbb{Z}/20$ is a corollary of that level's two mutually inverse buttons; on the other levels the buttons generate large non-commutative permutation groups.

Structure-layer checks. The Lean interface defines diagrams over finite quivers; cones over a quiver coincide with cones over the free category it generates, so identities and composites are not listed. Its names follow Mathlib's `Cone`, `Cocone`, `IsLimit` and `IsColimit`, but it is implemented in Lean core. The final check compiles every module, requires each object type with constraints and each case table to have a limit or colimit proof, requires every transformation to have an entry in the registry whose statement compiles, forbids the computational layer from importing the structure layer, forbids `sorry`, `admit` and `native_decide`, and audits the axioms of every constant in the project, not only those with a `#print axioms` line.

A.4 Lean encoding, certificates and transfer. Formalization starts from the consolidated objects and rules. AWM encodes object types as Lean `structure`, `inductive` or quotient declarations and primitive rules as executable `def` definitions. The presentations, case tables and transformation statements written during Structure & Simplify to check each step (§A.3) form an auxiliary layer that the model does not import. Consequences derivable from these definitions are proved as `theorem` statements in the derived layer. The agent assembles the modeled transition and goal rules in `Step.lean`, whose `step` function dispatches to the named rule definitions and returns a

successor or a terminal outcome for actions admitted by the model's legality conditions. For example, in `lp85`, a click either leaves the colors unchanged or applies a button's permutation, after which the goal rule determines whether the level is won; color conservation is a theorem derived from these definitions. This shared transition function is used both to replay recorded trajectories and to reason about unobserved action sequences. In `lp85`, the two budget-display rules remain in Python; Lean replays the tile dynamics and goal condition, while color conservation follows from the permutation rule (Figure 7).

On this basis, $\mathrm{everWon}(w, \bar{a})$ holds when some legal prefix of $\bar{a}$ reaches a winning outcome. The two statements of §2.4 are

$$\mathrm{Winnable}(w) :\Leftrightarrow \exists\, \bar{a}.\ \mathrm{everWon}(w, \bar{a}) = \mathsf{true}, \qquad \mathrm{Unsolvable}(w) :\Leftrightarrow \forall\, \bar{a}.\ \mathrm{everWon}(w, \bar{a}) = \mathsf{false}.$$

For level ℓ, the initial world w_0^ℓ is parsed from that level's first screen in the certified run, and the certified actions are $\bar{a}^\ell$. The reachability proof establishes $\mathrm{traceOk}(w_0^\ell, \bar{a}^\ell)$, i.e. that every computed state agrees with the recorded frame on the components the formal model represents, and $\mathrm{Winnable}(w_0^\ell)$ with $\bar{a}^\ell$ as witness. An impossibility proof establishes $\mathrm{Unsolvable}$ within a budget through a potential satisfying Eq. (3). Each theorem is closed by kernel reduction, its axioms lie within $\{$`propext`, `Classical.choice`, `Quot.sound`$\}$, and no proof uses `native_decide`; compiled with Lean 4.34.0, the published `Game.lean` of every game compiles and its `consistent` and `faithful` theorems depend only on `propext`.

The coverage definition in §2.4 asks the model to reproduce only the environment's optimal winning sequences under a fixed map h, rather than every transition as in an over-approximation (Aineto & Scala, 2024; Clarke et al., 1994). Let $\widehat{G}$ be the model's goal set.

Theorem A.1 (Transfer under optimal coverage)**.** *Suppose $\widehat{\mathcal{M}}$ covers the optimal winning sequences of $\mathcal{E}$ via h, and its initial state is $h(s_0)$. If $\mathrm{Unsolvable}_{\widehat{\mathcal{M}}}\big(h(s_0)\big)$, then no legal action sequence from s_0 reaches G.*

Proof. Suppose some legal sequence from s_0 reaches G. Then the task can be won, so it has an optimal winning sequence $\bar{a}$. By coverage, $\bar{a}$ is legal in $\widehat{\mathcal{M}}$ from $h(s_0)$, passes through the images of the real states, and ends in $\widehat{G}$. This is a model win, contradicting $\mathrm{Unsolvable}_{\widehat{\mathcal{M}}}(h(s_0))$. □

Full transition coverage is a special case: a model that reproduces every legal transition, with preserved goals and budget feasibility, reproduces every optimal winning sequence. For budgeted tasks, the optimal winning sequences are those of least cost, and the lemmas below give sufficient conditions that can be checked on engine-derived graphs.

For a fixed budget, let $\mathrm{Tr}_X(B)$ denote legal executions from the specified initial state that respect all budget checks. The model budget $\widehat{B}$ may use explicitly translated cost units.

Lemma A.2 (Budget-local execution transfer)**.** *If a map $\kappa : \mathrm{Tr}_{\mathcal{E}}(B) \to \mathrm{Tr}_{\widehat{\mathcal{M}}}(\widehat{B})$ preserves winning outcomes, model impossibility at $\widehat{B}$ implies environment impossibility at B.*

Proof. A real win would map to a budget-feasible model win, contradicting model impossibility. □

Unlike full coverage, this condition permits macro actions, stuttering and compressed detours. Goals, one-use resources and intermediate budget guards must still be preserved; final net cost alone is insufficient.

Lemma A.3 (Budget-local invariant transfer)**.** *Let D_B contain all real configurations reachable within budget B from the specified initial state, including winning terminals. If a certificate predicate I_B holds initially, is preserved by every budget-admissible real transition in D_B, and excludes every real goal in D_B, then the environment has no win at budget B.*

Proof. Induction establishes I_B at every feasible prefix; a winning prefix contradicts goal exclusion. □

For a potential H, summing its checked transition inequalities and terminal condition gives $H(q_0) \leq \overline{C}$ for any winning execution, where $\overline{C}$ is its total in the proof's cost units. A real budget-B win is excluded once cost accounting gives $\overline{C} \leq \widehat{B} < H(q_0)$. Thus only the certified budget obstruction, not every intermediate modeled state, needs to be preserved.

A partial certificate may be completed, but the required initial bound and all transition and winning-boundary inequalities must then be verified, including any revised existing values. Appendix C.5 gives the engine checks.

B Twenty-five games

This appendix gives the per-game data behind §3, Table 1 and Table 2, the test of frozen rules on unseen transitions, and the `lp85` proof sizes of §2.5.

B.1 Per-game results. Table 4 gives AWM's results and model for each game; Table 5 gives the official scores of all systems.

Table 4: AWM on each game. *Score*: official score of the certified run. *Cert.*: actions in the certified clears, which the score counts. *Sent*: every action AWM sent to the engine, including uncertified sessions, failed attempts and resets. *Human*: actions of the human baseline. *Concepts*: concepts written during play, with those withdrawn in brackets. *Rules (Lean)*: rules translated into Lean. *Replayed*: levels whose certified winning run Lean replays from the first screen to the modeled goal.

Game	Lv.	Score	Cert.	Sent	Human	Concepts	Obj. types	Rules	Rules (Lean)	Replayed
ar25	8	100.0	253	253	748	17 (3)	6	8	5	8/8
bp35	9	100.0	374	374	651	26 (4)	3	11	10	9/9
cd82	6	100.0	70	661	171	44 (13)	7	11	10	6/6
cn04	6	100.0	261	745	789	22 (1)	4	9	8	6/6
dc22	6	100.0	1,037	2,664	1,228	48 (6)	8	17	14	6/6
ft09	6	100.0	80	168	208	15 (6)	6	7	4	6/6
g50t	7	100.0	288	811	879	20 (1)	10	13	11	7/7
ka59	7	97.6	421	4,061	730	22 (8)	9	11	10	7/7
lf52	10	100.0	905	905	1,339	47 (6)	12	11	10	10/10
lp85	8	100.0	79	200	388	23 (4)	6	6	4	8/8
ls20	7	100.0	368	1,467	776	22 (3)	12	21	18	7/7
m0r0	6	100.0	215	486	1,107	26 (5)	8	9	8	6/6
r11l	6	100.0	81	184	233	26 (4)	6	9	8	6/6
re86	8	100.0	547	2,153	1,255	40 (4)	7	12	8	8/8
s5i5	8	99.3	461	2,618	638	49 (9)	5	19	17	8/8
sb26	8	100.0	124	424	213	20 (1)	6	16	12	8/8
sc25	6	100.0	146	401	350	27 (1)	5	18	11	6/6
sk48	8	100.0	263	1,407	1,070	39 (8)	6	7	4	8/8
sp80	6	100.0	138	371	518	58 (23)	3	10	6	6/6
su15	9	100.0	90	904	361	37 (15)	5	16	12	9/9
tn36	7	100.0	125	292	317	33 (9)	5	7	4	7/7
tr87	6	100.0	119	430	414	13 (3)	6	7	4	6/6
tu93	9	100.0	187	566	462	17 (0)	8	10	8	9/9
vc33	7	100.0	167	538	447	14 (0)	9	10	7	7/7
wa30	9	100.0	649	1,521	1,843	37 (6)	8	8	7	9/9
Total	183	99.9	7,448	24,604	17,135	742 (143)	170	283	220	183/183

B.2 Frozen rules on unseen transitions. We evaluate whether learned rules generalize to unseen screens and detect changes in game mechanics. For `lp85` and `ft09`, we freeze the concepts recorded during play (19 and 9, respectively) and the rules after Structure & Simplify (6 and 7). We collect transitions from random presses (seeds 0–9), retaining only states whose screens never appeared in the recorded gameplay.

Each game is evaluated on new trajectories from the original levels (T1) and two modified levels, whose winnability is verified by exhaustive search. In `lp85`, the modifications preserve the mechanics but change the layout: U moves one goal tile, making level 6 unwinnable, while D adds another swap that restores winnability. In `ft09`, the modifications change the hidden click mechanic: FU replaces level 2's click pattern with a plus shape, making it unwinnable, while FD adds a one-pixel change that restores winnability.

Table 5: Official scores on the 25 public ARC-AGI-3 games. Baselines are the official ARC Prize results: Claude Opus 5 (High) and GPT-6 Astra (Max) under the standard harness, which provides no code execution, and GPT-6 Astra (Max) under its provider-adapter harness. AWM is scored on its certified run. Rodionov's agent from its ARC Prize scorecard (Rodionov, 2026b); Memento 3 as reported in (Zhao et al., 2026).

		Official score (max 100)					
Game	Lv.	Claude Opus 5	GPT-6 Astra	GPT-6 Astra (adapter)	Rodionov (2026b)	Memento 3 (Zhao et al., 2026)	**AWM (ours)**
ar25	8	100.0	100.0	100.0	100.0	100.0	**100.0**
bp35	9	0.0	2.2	100.0	100.0	100.0	**100.0**
cd82	6	0.6	82.9	100.0	100.0	100.0	**100.0**
cn04	6	47.6	100.0	100.0	100.0	100.0	**100.0**
dc22	6	44.8	100.0	100.0	100.0	100.0	**100.0**
ft09	6	100.0	100.0	100.0	100.0	100.0	**100.0**
g50t	7	0.0	0.0	100.0	100.0	100.0	**100.0**
ka59	7	10.4	75.0	100.0	100.0	100.0	**97.6**
lf52	10	1.8	10.9	100.0	100.0	100.0	**100.0**
lp85	8	100.0	100.0	100.0	100.0	100.0	**100.0**
ls20	7	5.4	83.5	100.0	100.0	100.0	**100.0**
m0r0	6	28.6	71.4	100.0	100.0	100.0	**100.0**
r11l	6	100.0	100.0	100.0	100.0	100.0	**100.0**
re86	8	58.3	77.8	100.0	100.0	100.0	**100.0**
s5i5	8	100.0	100.0	100.0	100.0	100.0	**99.3**
sb26	8	77.8	100.0	100.0	100.0	100.0	**100.0**
sc25	6	0.0	47.6	100.0	84.2	100.0	**100.0**
sk48	8	8.3	19.7	100.0	100.0	100.0	**100.0**
sp80	6	56.3	100.0	100.0	100.0	100.0	**100.0**
su15	9	10.4	27.4	100.0	100.0	100.0	**100.0**
tn36	7	6.0	53.6	100.0	90.1	100.0	**100.0**
tr87	6	47.6	100.0	100.0	100.0	100.0	**100.0**
tu93	9	3.6	3.1	100.0	100.0	100.0	**100.0**
vc33	7	98.8	73.3	100.0	100.0	100.0	**100.0**
wa30	9	10.6	80.0	100.0	100.0	100.0	**100.0**
Mean		40.7	68.3	100.0	99.0	100.0	**99.9**
Games at 100		5	10	25	23	25	**23**

A rule set accepts a transition if no rule judges it false, and rejects it if at least one rule judges it false. We also generate invalid transitions using five corruption operators to test whether the rules can detect inconsistencies. Table 6 reports the results before and after Structure & Simplify. The frozen rules generalize to modified layouts in `lp85`, while detecting all tested transitions as inconsistent after the mechanic changes in `ft09`.

Table 6: Frozen-rule evaluation on unseen transitions, before / after Structure & Simplify (%). `lp85` U/D test generalization to modified layouts, while `ft09` FU/FD test detection of changed mechanics. "Wrong rejected" reports the range over the four corruption operators for which both rule sets can read both frames. The 0% acceptance on FU/FD indicates that all tested transitions violate the frozen rules: the "nothing else changes" rule detects clicks changing 110 or 146 cells instead of 38.

Set	n	Transitions accepted	Wrong rejected	Next frame predicted
lp85 T1 (new trajectories)	2,073	100 / 100	99.5–100 / 99.5–100	99.95 / 99.95
lp85 tile swap U	2,080	100 / 100	100 / 100	100 / 100
lp85 tile swap D	2,080	100 / 100	100 / 100	100 / 100
ft09 T1 (new trajectories)	2,026	99.0 / 100	99.0–100 / 100	—
ft09 pattern change FU	2,015	0 / 0	—	—
ft09 pattern change FD	2,016	0 / 0	—	—

`ls20`. On `ls20`, the concepts written during play accept 67–70% of valid transitions on unseen screens; after Structure & Simplify, the frozen rules accept 99.47–99.95% and reject 99.1–99.3% of corrupted transitions, against 71.8–100% for the concepts (Table 7). The gap comes from wall bumps: the concepts `energy_tick` and `energy_cost` charge energy for a blocked move only on the two levels where play recorded one, while the rule `energy_cost` charges every wall bump the level's step cost, which holds on screens the agent never saw.

Table 7: Frozen concepts and rules on two `ls20` test sets of previously unseen screens (%). T1: new trajectories on the original levels; T2V: level 1 with its switch moved one cell. Corrupted rejected: range over the four corruption operators for which both sides can read both frames.

		Valid accepted		Corrupted rejected	
Test set	n	Concepts	Rules	Concepts	Rules
T1	2,017	67.0	99.95	75.3–100	99.1–99.3
T2V	2,074	70.3	99.47	71.8–100	99.1–99.2

B.3 `lp85`: the complete model and proof size. Figure 7 shows the complete model of `lp85`. To evaluate the effect of Structure & Simplify on proof construction, we compare model A, the eleven concepts recorded during play, with model B, the single merged `press_permutes` rule. We ask Claude Opus 5.5 to establish the solvability (with a winning click sequence) or unsolvability (with a Lean proof) of 34 tasks under each model. Each task is run three times per model, yielding 204 runs in total, and every verdict is checked against exhaustive search on the engine.

Table 8 reports proof success and complexity. Model B succeeds in all 102 runs, compared with 99 for A, and generally requires fewer auxiliary lemmas, shorter proofs, and less Lean checking time.

No run produces an incorrect verdict: A has two timeouts and one run ending without an answer. On winnable instances, both models establish reachability using a winning click sequence and `decide`.

B.4 Checking a model-level proof against the engine. This check concerns the level-6 tile-swap variant U of Appendix B.2. We export its engine-derived, budget-independent reachable configuration graph as a Lean table, with 7,500 configurations and 52,500 transitions for the seven nontrivial button actions. A Lean proof checks each transition against the premises of the model-level unsolvability theorem, so the conclusion transfers to the exported graph. The proof uses only Lean's standard axioms. Extracting the graph from the game engine remains outside the Lean kernel's guarantee.

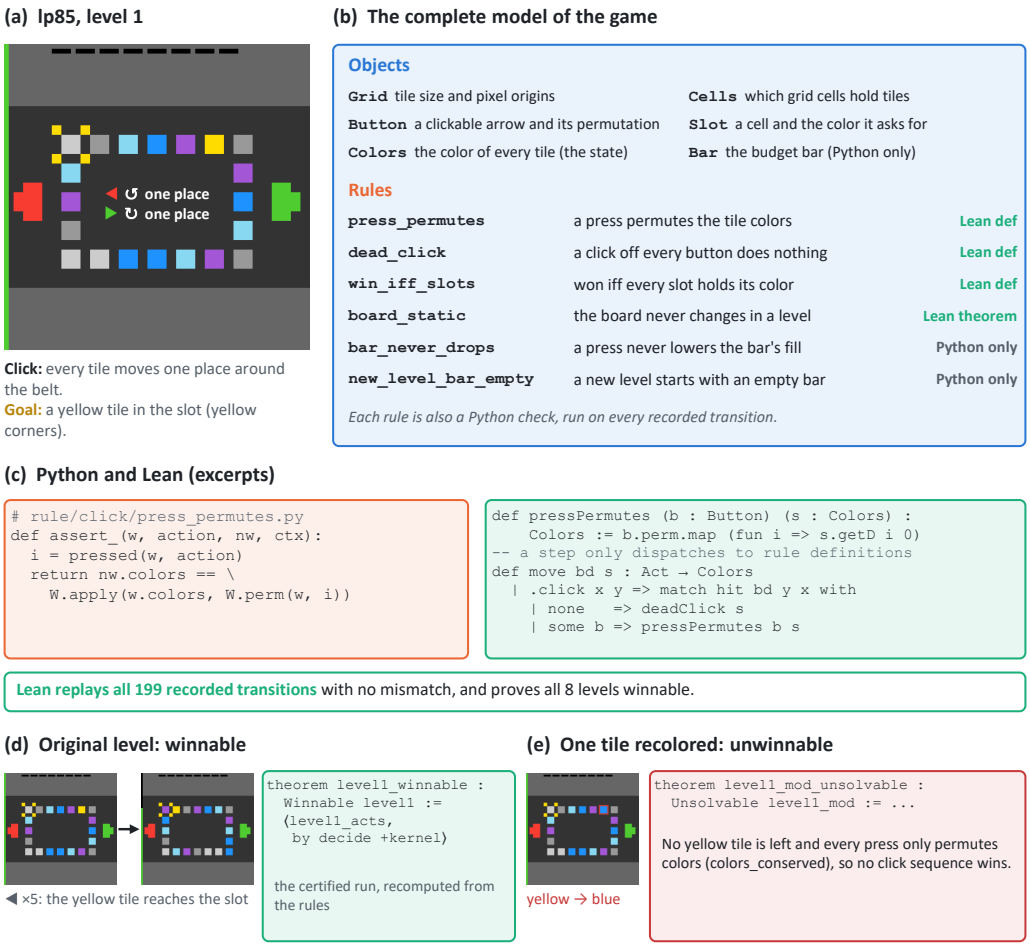

Figure 7: The complete algebraic world model of an ARC-AGI-3 game. (a) Level 1 of `lp85`: 20 colored tiles form a closed belt, and each click on an arrow moves every tile one place around the belt, counterclockwise for the left arrow and clockwise for the right; the level is won when a yellow tile stands in the slot marked by yellow corners. (b) All objects and rules of the model, with the form each takes in Lean; the two rules about the budget bar stay in Python. (c) Excerpts: one rule as its Python check, the same rule as a Lean definition, and the Lean step, which only dispatches to rule definitions. Lean replays every recorded transition of the game with no mismatch. (d) On the original level, Lean checks a winning witness, the certified run. (e) With the only yellow tile recolored, the conservation law of `press_permutes` shows that no click sequence wins. Both statements are theorems about the model.

Table 8: Proof success and complexity for 34 `lp85` tasks using the original concepts (A) or the merged rule (B). Verified counts successful runs; auxiliary lemmas, proof lines, and kernel checking time are medians over successful runs. The macro-average is the unweighted mean of the six group medians. All values are reported as A / B.

Instances	n	Verified A / B	Aux. lemmas A / B	Proof lines A / B	Kernel time (s) A / B
original levels (winnable)	24	24 / 24	0 / 0	4.5 / 4	1.00 / 0.83
goal colour removed (unwinnable)	24	24 / 24	6 / 3	75.5 / 28	0.90 / 0.80
level 6 swaps (U and D)	6	6 / 6	2 / 1.5	36.5 / 24	1.69 / 1.22
large layouts (winnable)	18	16 / 18	0 / 0	5.5 / 4	1.95 / 1.00
large layouts, altered (unwinnable)	18	17 / 18	12 / 4	117 / 49.5	2.82 / 2.41
ring spacing (unwinnable)	12	12 / 12	12 / 6.5	111 / 62.5	4.27 / 3.02
Macro avg. of medians	–	–	5.33 / 2.50	58.33 / 28.67	2.11 / 1.55
All runs	**102**	**99 / 102**	–	–	–

Figure 8: The three introductory levels and two sealed evaluation levels of `sc25`, rendered from the engine in the official palette. Panels (a–c) isolate resizing, teleportation and shooting; panels (d–e) combine them. The annotations show each objective and the main stages of a minimum-budget winning route. The sealed caps, 18 and 35, are one unit below the required 19 and 36.

C Sealed levels

The sealed tasks give a known winning-cost threshold against which to assess the reasoning in §4. A proof can be valid under a learned model even when that model misstates the game. We therefore examine the initial bounds and their underlying rules, then follow the corrections through renewed proofs and checks against the engine. The final studies test how proof dependencies help locate a faulty rule and what the verification procedure itself can detect.

C.1 Construction and engine ground truth. Each task is obtained by lowering a solvable level's budget to one unit below its minimum feasible cap $B^\star$. This gives both a winning witness at $B^\star$ and a known impossible instance at $B^\star - 1$ (Table 9).

Table 9: Reference thresholds for the sealed tasks. $B^\star$ is the minimum feasible cap; "gross" adds back the fuel refund of 10. Each witness wins at $B^\star$ and fails at $B^\star - 1$, while exhaustive search finds no winning route at the sealed cap from any opening facing. Route lengths count actions, while costs use the engine's counter units.

Level	Public cap	Sealed cap	$B^\star$ (gross)	Optimal route
4	35	18	19 (29)	27 actions
5	65	35	36 (46)	42 actions

`sc25` is a deterministic, top-down puzzle game in which a block must reach an exit through walls and gates. Direction commands set its facing and move it four pixels when large (4×4) or two when small (2×2); a blocked large move also tries a two-pixel step. Matching a pattern on the 3×3 panel casts a spell: resize, teleport to a size-dependent destination, or fire a straight ray at a target to remove its gates. Walls and gates stop the ray. Direction commands and panel clicks cost one unit, spell completion costs one more, and fuel refunds ten. The winning move is free; exceeding the cap ends the attempt. The right-edge bar displays the remaining budget.

We rearrange the public levels' wall masks and object placements while preserving the transition code, existing sprite definitions, spell patterns and cost rules. Before sealing, available spells and budget caps match the public levels. Figure 8 shows the resulting tasks. Levels 1–3 introduce shrinking through a narrow corridor, teleporting into an isolated exit chamber, and shooting to open a gated passage. Their caps are 50, 25 and 50; replayed witnesses use 15, 6 and 14 actions.

Level 4 combines a target at the end of a two-pixel-wide shaft, a gate across the right exit passage, and fuel. The large block's initial movement grid misses the shaft. A minimum-budget route shrinks and regrows over the fuel, obtaining the refund and changing its alignment, then lines up a shot and exits through the opened gate. Its 27 actions require a cap of 19 (gross cost 29). Level 5 has two target–gate groups, fuel, a narrow connector and separate teleport destinations for each size. A minimum-budget route shoots the first target, collects fuel, shrinks through the connector, regrows to shoot the second target, and teleports as a large block into the exit shaft. Its 42 actions require 36 (gross cost 46). These routes minimize the required budget, rather than the unconstrained number of actions.

The witnesses establish that caps 19 and 36 suffice. To establish that lower caps cannot win, we search the sealed engine before any agent run, covering all four opening facings and all 13 actions (four moves and nine panel cells). The reachable configurations number 905,587 on level 4 and 1,423,609 on level 5. For each configuration, the search retains the least counter value, since a lower counter leaves at least as much budget for every continuation. We audit all six counter updates and both cap checks in the source. To check for effective clicks omitted by the action set, we also scan the whole 64×64 screen on 30,001 sampled frame classes per level; clicks outside the nine panel cells change none of the 13 actions' outcomes. Editing the cap field changes only the budget, so the winning and sealed instances share the same dynamics.

C.2 Agents and protocol. The comparison tests whether agents can identify the budget obstruction and justify it from interaction. Three configurations play each sealed level, three runs each. AWM uses Claude Opus 5.5 with the AWM procedure, a generic Lean skeleton and tools for recording predictions and checking rules; it receives no game rules. Claude Opus 5.5 and GPT-6 Astra without AWM receive the task, game interface and ordinary coding-agent tools.

Each run has up to eight reset-delimited attempts, ending at game over or recorded abandonment. After each attempt, the interview in Appendix D records the agent's verdict and argument. The engine reference thresholds are withheld: arenas have opaque names, contain no ground-truth file, and begin with cross-run memory cleared. This keeps the agent's reported bound separate from the reference search used to evaluate it.

C.3 Proof validity and bound accuracy. The per-run results explain why a model proof and an accurate account of the game are scored separately. All six AWM runs produce closed Lean impossibility proofs, but two initially report overly high thresholds: 20 on level 4 and 47 on level 5. Their proofs certify the learned rules; the numerical errors reveal that those rules exclude cheaper behavior allowed by the engine. Table 10 also records the baseline verdicts, including an impossibility claim later withdrawn. These individual outcomes underlie Table 3 and motivate the repair round below.

First claim records the attempt of the earliest impossibility verdict, even if later withdrawn; it is not proof-completion time. *Bound* is the least winning cost reported before feedback, in the run's own units for runs without AWM. The reference $B^\star$ is 19 on level 4 and 36 on level 5. A bound counts as correct if it equals $B^\star$ or differs only by charging the otherwise free winning move: 37 on level 5, or 42 actions against a budget of 41 in `fbac96`. This accounting allowance applies to every agent.

Table 10: Per-run verdicts, reported bounds and model proofs for the 18 runs summarized in Table 3. "Correct" evaluates the reported bound against the engine; "Lean proof" records a closed, kernel-checked impossibility theorem under the run's learned model. The text specifies the timing and cost-accounting conventions.

Level	Agent	Run	First claim	Final verdict	Bound	Correct	Lean proof
4	AWM	`9ec8d4`	2	impossible	20	no	yes
4	AWM	`69fe87`	1	impossible	19	yes	yes
4	AWM	`2a3cc5`	2	impossible	19	yes	yes
4	Claude Opus 5.5	`2f1a1c`	—	undecided	—	no	no
4	Claude Opus 5.5	`d267df`	—	undecided	—	no	no
4	Claude Opus 5.5	`1dd6e6`	—	undecided	—	no	no
4	GPT-6 Astra	`44ec53`	8	undecided	22	no	no
4	GPT-6 Astra	`4d3b31`	8	impossible	22	no	no
4	GPT-6 Astra	`98893b`	—	undecided	—	no	no
5	AWM	`284967`	3	impossible	37	yes	yes
5	AWM	`4f4628`	2	impossible	36	yes	yes
5	AWM	`977714`	2	impossible	47	no	yes
5	Claude Opus 5.5	`339b0a`	4	impossible	47–49	no	no
5	Claude Opus 5.5	`202da3`	6	impossible	43	no	no
5	Claude Opus 5.5	`fbac96`	6	impossible	42 (budget 41)	yes	no
5	GPT-6 Astra	`2cdca5`	—	undecided	—	no	no
5	GPT-6 Astra	`3eb226`	—	undecided	—	no	no
5	GPT-6 Astra	`b3830e`	—	undecided	—	no	no

C.4 Repairing the rules and proving again. The two overestimated AWM bounds provide a test of whether an explicit model can be corrected after a valid proof has been obtained. After the eight-attempt round, those runs and the four baseline runs with wrong bounds receive one piece of feedback: the engine permits a cheaper win than their analysis allows. The message reveals no rule, cost or route. AWM additionally receives a self-audit instruction to relax rules consistently with the record, test disagreements in the game, revise and prove again; the baseline message states the same discrepancy logic in plain language. Each run gets at most four further attempts.

The corrections change specific premises of the proofs (Table 11). On level 4, growing onto fuel collects it, reducing AWM's bound from 20 to 19; Figure 9 shows the missed interaction and the cheaper route. On level 5, terrain rather than limited range blocks a shot, reducing the bound from 47 to 37 under the run's convention of charging the winning move, equivalent to the engine's 36. Both repairs produce new closed Lean proofs. Astra corrects both numerical bounds without producing a formal proof, while Claude Opus 5.5 corrects neither. Thus numerical revision and renewed formal certification remain distinct outcomes of the same feedback.

Table 11: Outcomes after feedback that a cheaper win exists. The revised-rule column identifies the premise changed by AWM before proving again. Bounds of runs without AWM remain in their own units, with the budget they state. Attempts count additional attempts in the repair round.

Run	Level	Attempts	Rule revised	Bound after	Lean proof
AWM `9ec8d4`	4	4	growing onto fuel collects it	19	yes
AWM `977714`	5	4	terrain, not range, stops the shot	37	yes
GPT-6 Astra `44ec53`, `4d3b31`	4	$\leq$4	—	19 (both)	no
Claude Opus 5.5 `339b0a`, `202da3`	5	$\leq$4	—	42 (budget $\sim$40), 68 (budget 64)	no

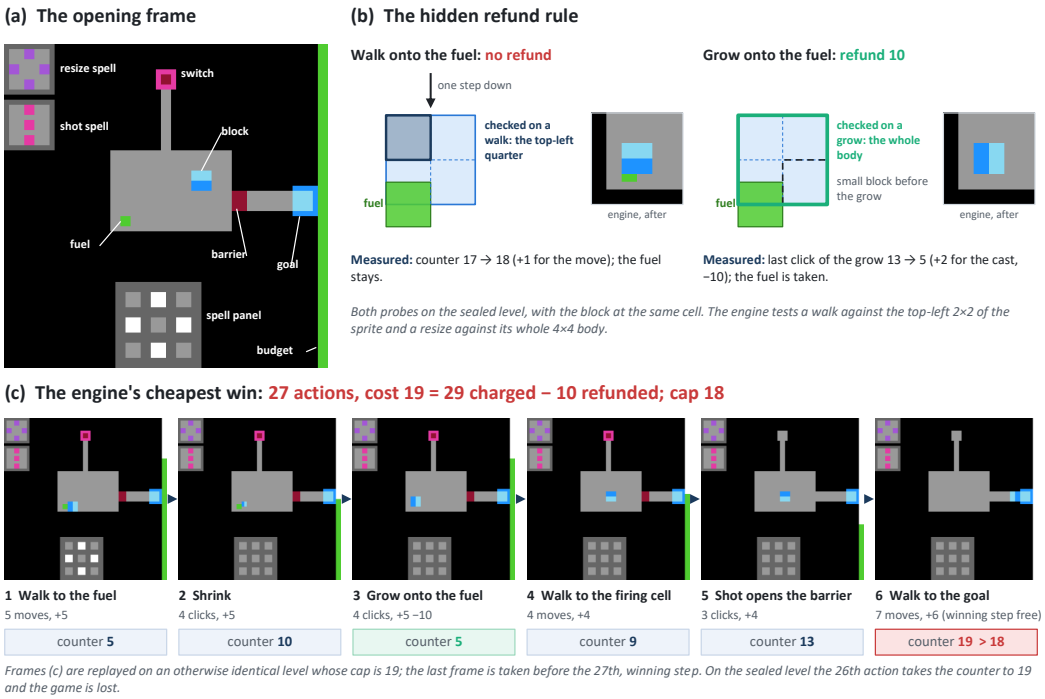

Figure 9: Why the level-4 model needed revision. (a) The opening. (b) Walking checks only the large block's top-left quarter for fuel, whereas growing checks its full footprint and can collect fuel under the bottom-left quarter, refunding 10. Missing this distinction led one AWM run to report 20. (c) The cheaper witness uses 27 actions and requires 19. It is replayed at cap 19; at the sealed cap 18, action 26 ends the game. All frames come from the engine.

C.5 Transferring budget certificates to the engine. We check each model proof's invariant or potential certificate on engine-generated transitions, using completed `2a3cc5` V4, feedback-repaired `9ec8d4` and `977714`, and the other runs' final models. The initial condition, preservation on real transitions and exclusion of real wins are checked as required by Lemma A.3. Five proofs extend to the games through the arguments below (Table 12).

Domain and budget accounting. The unchanged engine supplies physical transitions for four directions and nine panel cells. At cap 18, L4 has 656,715 reachable physical–panel configurations and 8,537,295 atomic branches; monotonicity permits keeping only each configuration's minimum counter. L5 has 4,842 physical configurations and 33,894 spell/move edges, including winning branches. Let q be the physical state, c the real counter, m the lit panel cells and $u = c - |m|$ the counter after crediting prepaid clicks. Completion clears the panel; extra toggles cannot reduce this accounting. Intermediate budget checks, clipped refunds and the free winning move are checked explicitly.

Table 12: Budget-local certificate audit at the fixed opening and sealed caps, 18 on L4 and 35 on L5. Pass includes the listed post-hoc treatment; it does not assert full coverage.

Level	Run / snapshot	Transfer argument	Result
4	`9ec8d4`, repaired	Original NN/MM values; panel-progress lifting	Pass
4	`69fe87`	Original V and domain; panel credit	Pass
4	`2a3cc5`, V4	Original arrival-cost closure invariant	Fails
5	`284967`	Extend potential; bridge macro prices and fuel	Pass
5	`4f4628`	Extend weighted certificate; translate budget	Pass
5	`977714`, repaired	Extend certificate using real growth and costs	Pass

L4: unchanged values, lifted invariants. For repaired `9ec8d4`, we check

$$\mathrm{Dom}(q) \wedge c + NN(q,m) \geq 19 \wedge \big(f \Rightarrow c + MM(q,m) \geq 29\big),$$

where f means fuel remains and the original NN/MM definitions credit missing spell clicks. For `69fe87`, we check $\mathrm{valid}(q) \wedge u + V(q) > 18$. Both hold throughout the cap-18 domain, with all atomic and terminal branches checked.

L5: extended certificates and cost bridges. Each successor is generated by the engine, rather than taken from the learned step function. Missing table values are completed, and existing values revised where needed, so that the potential inequalities of Eq. (3) and the winning boundaries hold on every real edge while retaining the required initial bound. The cases below use each certificate's own cost units. Let N count primitive actions through the win, K spell completions, and $p \in \{0,1\}$ fuel pickups.

For `284967`, a blocked two-pixel move can turn in place, whereas the model advances one pixel; the new position–facing combinations are added to the certificate domain. Its original prices are $(1,4,5,3,4)$ for move, shrink, grow, fire and teleport, with refund 8. The extended potential gives $M \geq 36$, where M is a winning trace's total in these units. Writing n_s, n_f for shrink and fire completions, real winning expenditure satisfies

$$c_{\mathrm{win}} \geq M + n_s + n_f - 2p - 1.$$

An auxiliary certificate gives $n_s + n_f - 2p \geq 1$, hence $c_{\mathrm{win}} \geq 36 > 35$. The -1 accounts for the free winning move; refund clipping and extra clicks only increase real expenditure.

For `4f4628`, destroying a target frees its former footprint; the model had opened only the associated gate. Completing these real successors retains the weighted bound $W = 9N + 10K - 90p \geq 338$. A real cap-35 win requires $N + K - 10p - 1 \leq 35$ and $N \geq 3K$. Since $p \leq 1$, $K \leq 11$ and $W \leq 324 + K \leq 335$, a contradiction.

For repaired `977714`, growth is completed with the engine's first-fitting search over nine offsets in $\{-2,-1,0\}^2$, instead of only $(-2,-2)$ and $(0,-2)$. The extended potential retains initial value 37 in real units with the winning move charged. The invariant $u + H(q) \geq 37$, together with $H(q) \leq 1$ at a real winning edge, requires counter at least 36.

Allowing or forbidding entry into destroyed-target footprints, or suppressing additional growth placements, leaves the engine's minimum cap at 36, although local continuation costs can change. The completed certificates include these outcomes, preserve the required initial lower bounds, and satisfy every transition inequality and winning boundary after cost translation; hence the local changes do not invalidate the impossibility arguments.

Why `2a3cc5` V4 fails. Its arrival-counter invariant requires $g(q) \leq \max(0,u)$. After `LLLLDD`, the real block at $(33,21)$ retains fuel and has counter 6, but the table gives $g = 17$: V4 incorrectly collects fuel. After `LLLDDSS`, with S a resize spell, growth collects fuel and the real counter falls from 10 to 5; V4 misses the refund, and the table gives the real successor $g = 9$. Thus both $17 \leq 6$ and $9 \leq 5$ fail, as checked with the original Lean definitions. Its pre-payment guard also rejects a real surviving refund move from 18 to 9. This audit leaves V4 unreconciled.

These are post-hoc certificate audits, not additional agent-produced proofs. Archived Lean definitions supply the original values; offline programs check graph extraction, reconciliation and engine

correspondence, outside an end-to-end Lean verification of the engine. Their fixed-opening scope is distinct from the four-opening ground-truth search of §C.1.

C.6 Using proof dependencies to locate a faulty rule. The repairs above identify concrete rules that blocked cheaper solutions. To measure whether proof dependencies help find such rules, we inject a known single-rule error into a reference model that matches the engine under the abstractions of §C.1. The main sample comprises the 14 level-4 and 7 level-5 injections that change the model's least winning cost, each run with seeds 1–3.

Claude Opus 5.5 receives the faulty model, its proof and a service for probing the sealed engine. C1 adds a per-rule evidence table; C2 adds the proof's rule dependencies to the same table, without naming the faulty rule. R provides a reference condition in which that rule is named. C0 probes randomly without an agent, and EX searches the engine exhaustively. A discovery requires an observed transition, within the faulty rule's scope, on which the engine and model disagree. It is counted in engine actions to the first such observation, with budgets of 50 actions on level 4 and 5,000 on level 5. Naming a rule in the final report is recorded separately from exposing its error through interaction.

Table 13 shows where the added dependency information helps. On level 4, C2 discovers the conflict in 29 of 42 runs, compared with 16 of 42 for C1 and 30 of 42 when the faulty rule is named. Level 5 remains harder: discovery is 3 of 21 for C2 and 1 of 21 for C1. Most of the improvement in the main sample therefore comes from level 4; the aggregate difference and error-free controls are reported with the table.

Table 13: Counterexample discovery with injected rule errors: 14 injections on level 4 and 7 on level 5, each with three seeds. *Found*: discovery within the action budget. *Median*: Kaplan–Meier median actions to discovery, censored at the budget. *Named*: the conflict log or final model identifies the faulty rule. EX has no action budget. Across the 63 sample–seed pairs, C2's discovery rate exceeds C1's by 0.238 (sample-clustered bootstrap 95% interval $[0.079, 0.397]$). On 12 error-free controls, C2 raises 0.00 false alarms per run and C1 0.33.

Arm	L4 found	L4 median	L5 found	L5 median	Found (L4 + L5)	Named	Tokens (median)
C0 random	12/42	> 50	0/21	> 5000	12/63	—	—
C1 per-rule evidence	16/42	> 50	1/21	> 5000	17/63	35/63	2.35M
C2 proof dependencies	29/42	27	3/21	> 5000	32/63	50/63	3.29M
R faulty rule named	30/42	26	5/21	> 5000	35/63	61/63	1.98M
EX exhaustive	14/14	136,852	1/7	—	15/21	—	—

The level-5 comparison reported with 42 paired runs per arm uses the same 5,000-action discovery criterion: C2 finds the conflict in 8 of 42 runs and C1 in 5 of 42 (Table 14). These results retain their own denominator alongside the 21-run level-5 results above.

Table 14: The level-5 comparison reported with 42 paired runs per arm, using the same discovery criterion and 5,000-action budget. Its denominator is reported separately from the 21-run level-5 sample in Table 13.

Arm	L5 found	Discovery rate
C1 per-rule evidence	5/42	11.9%
C2 proof dependencies	8/42	19.0%

C.7 What the verification procedure checks. A checked proof establishes that a conclusion follows from its formal definitions and premises; correspondence with the game is assessed separately. To distinguish these guarantees, we introduce 21 mutations into each of three models: the sealed-level reference models (2,675 and 3,502 states) and the merged `lp85` model (869 states). Mutations affect certificate generation (G), model definitions (M1–M6), one of the Python or Lean representations (M7), or theorem statements (S1–S7). We compare a Python search for the least winning budget (V1), a Python checker of certificate conditions (V2), and Lean proof checking with an additional statement audit (V3).

Both V2 and V3 reject all 12 invalid certificates, including nine whose conclusions are false in the engine (Table 15). V3 additionally detects the tested Python–Lean mismatches (3/3) and reversed bounds (3/3); its separate statement audit flags six vacuous-premise or empty-start-set cases. These six theorems may still compile: the Lean kernel checks deductions from stated premises, not whether those premises are meaningful for the game.

None of the routes detects the injected model-definition errors (M1–M6) or the wrong-goal and model-version statements (S3, S7). The test therefore isolates what the verification pipeline checks beyond certificate validity, while also showing why interaction and engine-level checks remain necessary to connect the formal specification to the game.

Table 15: Mutation detection across three models. V1 searches in Python; V2 checks certificates in Python; V3 combines Lean proof checking with a separate statement audit. Vacuous premises and empty start sets are flagged by the audit, not rejected by the kernel. "n/a" denotes a mutation outside a route's inputs; undetected model and statement errors delimit the checks' scope.

Mutant	V1 Python search	V2 Python checker	V3 Lean + audit
generator error: invalid certificate rejected	n/a	12/12	12/12
of which the conclusion is false on the engine	n/a	9/9	9/9
model error (M1–M6) rejected	none	none	none
Python and Lean versions differ (M7) rejected	0/3	0/3	3/3
vacuous premises or empty start set (S1, S2) reported	0/6	0/6	6/6
reversed bound (S4) rejected	n/a	0/3	3/3
wrong goal set, other model version (S3, S7) rejected	none	none	none

D Interview prompts

This appendix reproduces verbatim the interview of Appendix C.2. After each attempt the harness sends questions 1–3, the formalization questions (AWM runs only) and the verdict lines, in the same session as the play; the level is then reset with the retry message. After the last attempt the final interview replaces questions 1–3 and keeps the same verdict lines.

Questions 1–3, every run.

```
That attempt failed. The game will reset this level, and you may get another attempt.

Answer the following. Keep it to what it needs; do not pad.

1. What happened on the level you were on when the attempt ended, and what do you think made
   it fail? Explain in plain prose, as if to someone who did not watch you play.

2. Do you still see a way to beat this level?
   - If you do: say what it is, and what you would still need to check to be sure of it.
   - If you no longer do: say why not. State precisely what you are claiming, give the
     argument for it, and make clear what the argument rests on - which parts you established
     and which are assumptions you did not check. Someone else should be able to follow it and
     judge whether it holds. The form of the argument is yours to choose.
   - If you cannot tell either way, say so, and say what would settle it.

3. What will you do on the next attempt?
```

Formalization questions, AWM runs.

```
4. Before you answer this one, bring your formal system up to date. It is the thing you are
   building across attempts; the proof is downstream of it. Work in this order, in your
   working directory:

   a. Objects. What things are on the board, and what slots does each thing carry? For every
      object you keep, you should be able to name a pair of positions that stop being
      distinguishable if you drop it.
   b. Rules. Write each rule as a function: given a position and an action, what does it
      change? Each rule needs a domain you can actually execute.
```

```
   c. Simplify. Drop slots no rule reads. Merge rules that always decide alike. After this,
      your model's verdict on every frame you have observed must be unchanged.
   d. Formalize. Turn the objects and rules into Lean *definitions* - not theorems. Each rule
      ends up in exactly one of three states: a definition, a consequence of definitions, or
      explicitly marked "not in the model".

   If you already have a formal system from an earlier attempt, **amend it. Do not start a new
   one.** What you learned this attempt is a change to specific objects, rules or definitions;
   make those changes and leave the rest alone. A rewrite throws away the part of your work
   that was already right, and it destroys the record of what this attempt actually taught you.

5. Now prove your answer to question 2 **on that formal system**: that the level can be won,
   or that it cannot.

   - The proof must compile, and `#print axioms` on your main theorem must come back clean -
     no `sorryAx`, no `native_decide`. Report that output verbatim.
   - Say what your theorem actually quantifies over and what it assumes. If the proof needs a
     premise you did not establish, write it as an explicit hypothesis rather than burying it
     in a definition.
   - Name the rules from step 4b that your theorem leans on. If your Lean definitions carry
     something your rules do not - an extra assumption, a tighter domain, a step treated as
     atomic - say so here. A proof that is clean about the wrong system is worth nothing, and
     a model that is *stricter* than the game is the dangerous direction: "impossible in my
     model" does not transfer to the game.
   - Put the file in `lean/` and tell me its filename.
   - If you cannot complete the proof, say exactly where it breaks and what is missing. An
     honest "it does not go through, here is the gap" is worth more than a theorem that proves
     something weaker than it looks.

6. If this attempt taught you nothing that changes your formal system - no rule revised, no
   definition touched - do not invent a change. Say so instead, on its own line, in exactly
   this form, and then give the reason:

       FORMAL-UNCHANGED: <why this attempt added nothing to the model - what you tried, what
       you already knew, and what would have had to happen for the model to change>

   "No change, and here is why" is a real result. A change made only to look busy is not.
```

Verdict lines.

AWM runs.

```
End your reply with exactly these two lines, on lines of their own, with nothing after them:

VERDICT: winnable | impossible | impossible_in_model_proved | impossible_in_model | undecided
CONFIDENCE: <an integer from 0 to 100>

The two "in_model" words are for when the level cannot be won under the rules you yourself
measured, and the step from "those rules" to "the real game" is an assumption you are stating
rather than something you established. Say which assumption, in your answer to question 2.
Use `impossible_in_model_proved` only if you are also handing over a Lean proof of it that
compiles and whose `#print axioms` is clean - name the file. Otherwise use `impossible_in_model`.
Your proof will be compiled and its axioms checked, so do not claim `_proved` on a file that
does not go through; `impossible_in_model` with an honest account of the gap is worth more.

Answer in writing only. Do not send any more game actions.
```

Baseline runs.

```
End your reply with exactly these two lines, on lines of their own, with nothing after them:

VERDICT: winnable | impossible | undecided
CONFIDENCE: <an integer from 0 to 100>

Answer in writing only. Do not send any more game actions.
```

Retry message.

```
You have another attempt on that level.

`POST /reset` restarts the level you were on; the levels you already beat stay beaten, and the
level's own counters start over. Play it again, as far as you can. When the attempt ends, stop
and report as before.
```

Final interview, questions.

```
That attempt failed.

Answer the following in writing. Give each answer the length it needs; do not pad.

1. What happened on the level you were on, and what do you think made it fail?
   Explain in plain prose, as if to someone who did not watch you play.

2. Do you still see a way to beat this level?
   - If you do: give the line of play, and say what you checked it against.
   - If you no longer do: say why not. State precisely what you are claiming, give the
     argument for it, and make clear what the argument rests on - which parts you established
     and which are assumptions you did not check. Someone else should be able to follow it and
     judge whether it holds. The form of the argument is yours to choose.
   - If you cannot tell either way, say so, and say what would settle it.

3. Which rule did you newly work out across these attempts, and from which observation?

4. Which rule are you still unsure about, and unsure in what way?
```