DecoVer: A Decompose-and-Verify Neuro-Symbolic Framework for Embodied Task Planning with $\mathcal{BC}+$

Yixiang Jiang [*1] **Binqian Xu** [*1] **Xiangbo Shu** [1]

Abstract

Despite their remarkable general capabilities, Large Language Models (LLMs) struggle with the precise grounding required for embodied task planning. To bridge this gap, neuro-symbolic approaches have emerged, leveraging action languages like $\mathcal{BC}+$ for their formal expressiveness and reasoning flexibility. However, prior methods that naively couple LLMs with $\mathcal{BC}+$ typically depend on one-shot program generation, which is brittle in dynamic environments and prone to **sequential omission** and **causal inconsistency**. To address these limitations, we propose DecoVer, a Decompose-and-Verify neuro-symbolic framework that systematically adapts $\mathcal{BC}+$ to embodied task planning. Specifically, DecoVer employs a cascading decomposition strategy to partition complex knowledge into hierarchical subspaces and integrates a dual verification mechanism for syntactic and executable correctness. Extensive experiments demonstrate that DecoVer consistently outperforms LLM-based baselines across the majority of evaluation metrics, achieving a 12.9% success rate gain over the highly capable Gemini-3-Pro-Preview and a 60.9% improvement over GPT-5.1 on logically complex test cases.

1. Introduction

Embodied task planning aims to enable agents to achieve specified goals through situated interaction with physical or simulated environments (Wu et al., 2023; Wang et al., 2025). As a core capability of embodied agents, it requires reasoning over grounded objects, spatial relations, and action dynamics while coordinating sequences of goal-directed behaviors (Li et al., 2025b). Despite their strong generalization across diverse tasks, Large Language Models (LLMs) face significant challenges in embodied task planning. Most LLM-based planners (see Figure 1 (a)) generate action sequences end-to-end through implicit reasoning about environment dynamics and task constraints (Li et al., 2024). However, such probabilistic generation is weakly grounded in the physical state and lacks verification mechanisms. When tasks become long-horizon or involve complex logical dependencies, LLM planners fail to preserve global state consistency or satisfy constraints, resulting in **(i) sequential omission** and **(ii) causal inconsistency** in the plans.

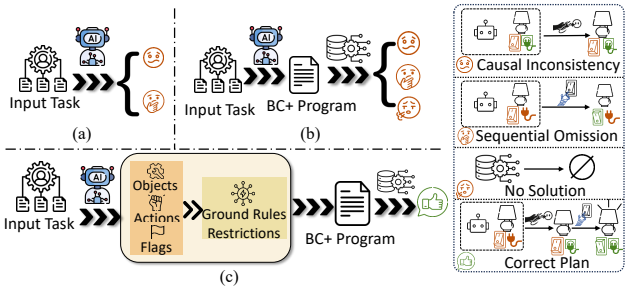

Figure 1. Comparison of task planning paradigms: (a) LLM-based planners, (b) Existing $\mathcal{BC}+$ neuro-symbolic methods, and (c) DecoVer (Ours). Compared to (a) and (b), DecoVer more effectively mitigates Causal Inconsistency, Sequential Omission, and no solution issues, significantly increasing correct plan yield.

Specifically, for **(i) sequential omission**, LLMs may skip necessary intermediate actions when preconditions are implicit or require multi-step causal reasoning, producing plans that appear plausible but are non-executable under the environment dynamics. For **(ii) causal inconsistency**, LLMs may lose track of state evolution over long action sequences, producing plans that contradict earlier action effects (e.g., executing inapplicable actions or assuming incompatible states). To address the above issues, neuro-symbolic approaches represent a promising direction (Gestrin et al., 2024; Ishay & Lee, 2025). They combine LLM reasoning with controllable and interpretable symbolic representations and execution, grounding planning in explicit states, preconditions, and causal effects, thereby helping to reduce typical errors in LLM-based planning (Fang et al., 2024). Existing

[*]Equal contribution [1]the School of Computer Science and Engineering, Nanjing University of Science and Technology. Correspondence to: Binqian Xu <xubinq11@gmail.com>, Xiangbo Shu <shuxb@njust.edu.cn>.

Proceedings of the 43rd International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026. Copyright 2026 by the author(s).

neuro-symbolic approaches for embodied task planning can be broadly categorized into three paradigms: Python-based programmatic methods (Ahn et al., 2026; Ling et al., 2025b), PDDL-based planning methods (Gestrin et al., 2024; Zhou et al., 2024), and action-language-based planning methods (Yang et al., 2023; Ishay & Lee, 2025). Among them, Python-based methods offer flexibility and generality, while Python's procedural nature limits its suitability for declarative constraint-based planning. Compared with PDDL-based methods that rely on rigid domain/problem specifications, action languages express preconditions, effects, and constraints in a declarative, causal-rule-oriented form, making modeling more direct and more expressive for complex dependencies, default reasoning, and global constraints. In particular, $\mathcal{BC}+$, as an action language (Babb & Lee, 2020) that integrates and encompasses the strengths of other action languages (e.g., $\mathcal{B}$, $\mathcal{C}$, $\mathcal{C}+$, and $\mathcal{BC}$), naturally emerges as a primary choice for our work.

Recently, LLM+AL (Ishay & Lee, 2025) made a first attempt to combine LLMs with the action language $\mathcal{BC}+$ for classical planning tasks with fixed rules and simple settings, such as MCP (McCarthy, 1998), yielding promising results. However, when directly applied to embodied task planning, richer object and interaction diversity and more complex dynamics limit its ability to reduce errors. Moreover, it introduces a new failure mode where LLM-generated $\mathcal{BC}+$ programs admit no solution under the solver (see Figure 1 (b)). After analyzing LLM+AL failure cases, we attribute these issues to two factors. First, it extracts the required knowledge in a single pass from the interactive environment, ***without explicit reasoning or decomposition***, making LLMs prone to hallucination. Second, converting this knowledge into rules is error-prone: ***syntactic mistakes*** are common, and inaccuracies in the extracted knowledge can compound into rule-level ***logical errors***.

To overcome these limitations, we propose a Decompose-and-Verify neuro-symbolic (DecoVer) framework grounded in the action language $\mathcal{BC}+$. As shown in Figure 1 (c), DecoVer employs a cascading strategy that decouples knowledge into three hierarchically dependent subspaces: Objects, Actions, and Flags. The framework sequentially identifies task-relevant Objects, infers Actions conditioned on these Objects, and finally resolves Flags based on the established Objects and Actions to capture intrinsic properties, dynamic states, and execution constraints. Leveraging these structured components, we formalize two rule categories: Ground Rules governing Action-induced Flag transitions, and Restrictions prohibiting Actions under specific constraints. To eliminate syntactic and logical errors, we present a dual-verification mechanism that integrates static code analysis with self-reflection to filter and rectify rules.

Our main contributions are summarized as follows: **(1) Novel neuro-symbolic paradigm.** We introduce the neuro-symbolic paradigm combining LLMs with the action language $\mathcal{BC}+$ to embodied task planning for the first time, bridging high-level semantic reasoning and grounded physical execution. **(2) Problem identification and empirical superiority.** We systematically analyze the critical bottlenecks of the direct LLM-$\mathcal{BC}+$ integration. By overcoming these inherent challenges, we demonstrate that DecoVer consistently outperforms baselines, achieving a 12.9% success rate gain over Gemini-3-Pro-Preview and a 60.9% gain over GPT-5.1 on logically complex cases. **(3) The DecoVer framework.** As our core technical contribution, we propose DecoVer. Featuring a Decompose-and-Verify mechanism, it effectively resolves the aforementioned bottlenecks to ensure the rigorous executability and logical integrity of synthesized programs.

2. Related Work

Large Language Models for Embodied Planning. Large Language Models (LLMs) have emerged as powerful engines for embodied reasoning. Early approaches (Yao et al., 2022; Brohan et al., 2023), demonstrated that LLMs could decompose high-level instructions into executable actions by interleaving reasoning traces with environmental feedback. Subsequent works have enhanced this via advanced prompting: Tree of Thoughts (Yao et al., 2023) enables non-linear path exploration, while RAP (Hao et al., 2023) incorporates Monte Carlo Tree Search to simulate future states. Systems like Voyager (Wang et al., 2024) leverage iterative code generation to facilitate exploration, and Reflexion (Shinn et al., 2023) introduces self-reflection mechanisms for error correction.

Yet, LLM-based planning remains bounded by the stochastic nature of the underlying model. As established in critical evaluations by Valmeekam et al. (Valmeekam et al., 2023), while LLMs can approximate reasoning,they lack a consistent modeling of the world; operating on statistical token likelihood rather than explicit state transition rules. DecoVer addresses this by shifting the LLM's role from a *planner* to a *formalizer*. Instead of requiring the model to simulate states which it does poorly, we leverage its semantic strength to construct a rigorous problem specification, offloading the burden of trajectory synthesis to a solver that guarantees physical consistency.

Neuro-Symbolic Planning and Action Languages. To combine the semantic flexibility of LLMs with formal correctness, current neuro-symbolic approaches for embodied task planning fall into three categories based on their symbolic forms: Python, PDDL, and Action Languages. Python-based frameworks such as (Ahn et al., 2026; Ling et al., 2025a; Chen et al., 2024) improve reliability through code generation and verification. However, the imperative nature

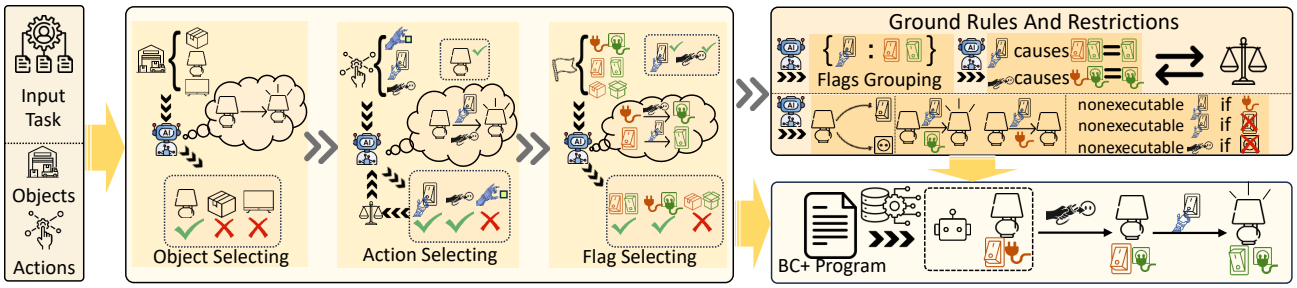

Figure 2. Overview of the proposed DecoVer framework. DecoVer comprises five decomposition stages: Object Selecting, Action Selecting, Flag Selecting, Ground Rule Generation, and Restriction Generation. Verification mechanisms are integrated into critical stages to perform corrections. Finally, the synthesized $\mathcal{BC}+$ program is executed to generate the action sequence.

of Python lacks the inherent capacity for declarative constraint expression found in professional planning languages. Conversely, PDDL-based methods (Liu et al., 2023; Gestrin et al., 2024; Zhou et al., 2024), employ LLMs to build formal domain and problem files for PDDL planners. While rigorous, PDDL often lacks the expressivity for indirect effects compared to action languages. Action Languages such as $\mathcal{BC}+$ (Ishay & Lee, 2025) offer the rigor of planning languages with greater flexibility than PDDL. Despite (Ishay & Lee, 2025) taking a significant first step by combining LLMs with $\mathcal{BC}+$, these approaches predominantly rely on direct translation strategies, where the model attempts to map the entire problem description into a formal specification. Such methods struggle in open-world embodied tasks where distracting objects and irrelevant details are present, a challenge further highlighted in (Choi et al., 2025).

We propose DecoVer to address these limitations. By decoupling the translation pipeline, DecoVer projects the environment onto minimal task-relevant subsets—a strategy conceptually echoing hierarchical abstractions in (Li et al., 2025a)—thereby filtering open-world distractors and bounding the hypothesis space for $\mathcal{BC}+$ program synthesis.

3. Methodology

3.1. Preliminaries and Framework Overview

Action Language $\mathcal{BC}+$. Action languages are formal languages designed to describe actions and their effects (Gelfond & Lifschitz, 1993; Babb & Lee, 2015; 2020; Gelfond & Lifschitz, 1998; Giunchiglia & Lifschitz, 1998; Giunchiglia et al., 2004; Lee et al., 2013). Among these, action language $\mathcal{BC}+$ (Babb & Lee, 2015; 2020) offers sufficient expressive power to integrate and encompass the core strengths of other action languages, such as $\mathcal{B}$ (Gelfond & Lifschitz, 1998), $\mathcal{C}$ (Giunchiglia & Lifschitz, 1998), $\mathcal{C}+$ (Giunchiglia et al., 2004), and $\mathcal{BC}$ (Lee et al., 2013). Structurally, A $\mathcal{BC}+$ program (Babb & Lee, 2015; 2020) comprises a signature and causal laws. The signature defines available symbols as constants, where constants are either fluent constants describing states or action constants representing actions. $\mathcal{BC}+$ formalism encompasses three categories of causal laws: static laws governing dependencies among fluents, action dynamic laws describing dependencies among concurrent actions, and fluent dynamic laws defining the effects of actions on states. Please refer to (Babb & Lee, 2015; 2020) for more details.

Overview. To address the common issues of sequential omission, causal inconsistency, and no solution when LLMs directly generate a $\mathcal{BC}+$ program, we propose DecoVer, a Decompose-and-Verify neuro-symbolic framework for task planning based on action language $\mathcal{BC}+$ (see Figure 2). In DecoVer, the generation of $\mathcal{BC}+$ programs is decomposed into five sequential stages: Object Selecting, Action Selecting, Flag Selecting, Ground Rules Generation, and Restrictions Generation. Specifically, Flag Selecting is subdivided into goal- and action-based selection processes; Ground Rules Generation takes the grouping of actions and Flags as input; while Restrictions Generation is further decoupled into generation steps targeting the Target Object and Environment Object. This approach embodies a cascading strategy, effectively decomposing knowledge into hierarchically dependent subspaces. To further address potential errors in Action Selecting and Ground Rules Generation, we implement a verification mechanism utilizing static code analysis and self-reflection. Finally, the solver executes the generated $\mathcal{BC}+$ program to output the action sequence.

Notations and Definitions. We formulate the task as a 6-tuple $\mathcal{T} = (\mathcal{O}, \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{F}, \mathcal{G})$, where $\mathcal{P}$ and $\mathcal{S}$ denote the properties and states of Objects $\mathcal{O}$, and $\mathcal{A}$ defines the Action space. To bridge raw observations with formal logic, the Flag library $\mathcal{F}$ encodes discrete environmental states. The goal $\mathcal{G}$ comprises state ($\mathcal{G}_s$), relational ($\mathcal{G}_r$), and action ($\mathcal{G}_a$) components, defining a conjunction of desired final states and relations. Thus, DecoVer aims to synthesize a $\mathcal{BC}+$ program whose execution yields an action sequence satisfying $\mathcal{G}$. In the first three stages, Object, Action, and Flag Selecting sequentially extract subsets $\mathcal{O}_s \subseteq \mathcal{O}$, $\mathcal{A}_s \subseteq \mathcal{A}$, and $\mathcal{F}_s \subseteq \mathcal{F}$ to construct the $\mathcal{BC}+$ signature. Subsequently, Ground Rules Generation and Restrictions Generation yield

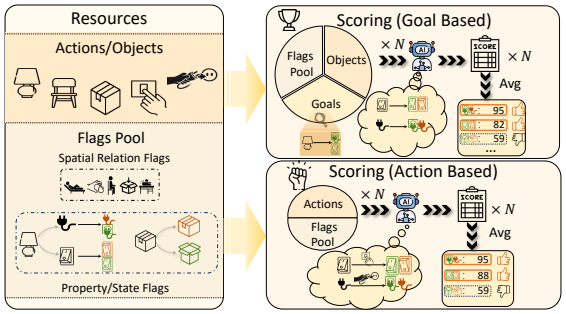

Figure 3. Flag Selecting.

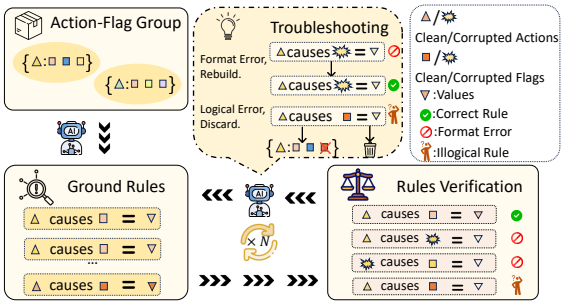

Figure 4. Ground Rules Generation and Verification.

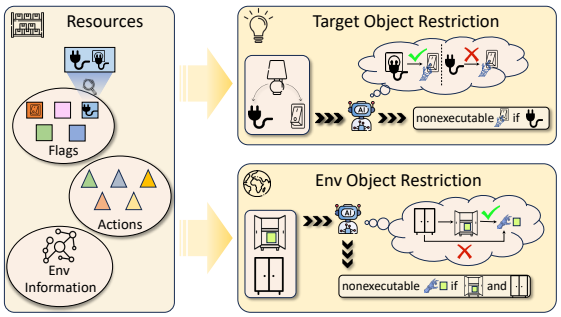

Figure 5. Restriction Generation.

Ground Rules $\mathcal{R}_c$ and Restrictions $\mathcal{R}_r$, respectively, which together constitute the causal laws. Here, $\mathcal{R}_c$ models dynamic state transitions triggered by Actions, while $\mathcal{R}_r$ prohibits Action execution under invalid states.

Scoring Schema. As the first three stages of DecoVer share a common scoring schema, distinguished only by their criteria sets, we formally define this mechanism before detailing the five-stage pipeline. Let $\mathcal{X} \in \{\mathcal{O}, \mathcal{A}, \mathcal{F}\}$ denote a candidate set of Objects, Actions, or Flags. We define an ordered list of selection criteria $\{C_1, ..., C_m, ..., C_M\}$ ranked by priority, where each C_m maps to a score interval $[l_m, r_m]$ ($l_m > r_{m+1}$). For an element $x \in \mathcal{X}$, the score $s(x)$ is determined by the highest-priority criterion it satisfies:

$$s(x) \in [l_{\hat{m}}, r_{\hat{m}}], \quad \hat{m} = \min_{m \in M} \{m \mid x \models C_m\}. \quad (1)$$

Here, $x \models C_m$ denotes that element x satisfies the corresponding criterion. To mitigate LLM stochasticity, we compute the final score as the mean $\bar{s}(x)$ over multiple independent scoring rounds, retaining elements where $\bar{s}(x) > \tau$.

3.2. Object Selecting

In the first stage of DecoVer, we perform Object Selecting based on the aforementioned scoring schema. The ordered list of selection criteria $\{C_1, C_2, C_3, C_4\}$ is defined as follows: (i) C_1: explicit mention in state goals $\mathcal{G}_s$; (ii) C_2: participation in relational goals $\mathcal{G}_r$; (iii) C_3: status as targets in action goals $\mathcal{G}_a$; and (iv) C_4: enabling objects for other objects. For instance, a closed container holding a goal-related object satisfies C_4. The absence of such objects would result in an incorrect environment model where the physical context of the target objects is undefined.

3.3. Action Selecting

Upon completion of Object Selecting, we utilize $\mathcal{O}_s$ and $\mathcal{G}$ to extract the Action subset $\mathcal{A}_s \subseteq \mathcal{A}$ according to the criteria $\{C_1, ..., C_4\}$ below: (i) achieve state goals $\mathcal{G}_s$; (ii) builds required relational goals $\mathcal{G}_r$; (iii) directly mentioned in action goals $\mathcal{G}_a$; and (iv): satisfies intermediate states.

Verification. To mitigate action omission, we employ a verification step where an independent LLM attempts to generate a plan satisfying $\mathcal{G}$ using only $\mathcal{A}_s$. If LLM encounters an unsatisfiable intermediate state, it returns a hint entailing the situation. This hint is used as additional input in a restarted scoring round to retrieve the missing action.

3.4. Flag Selecting

Based on $\mathcal{O}_s$ and $\mathcal{A}_s$, we construct the Flag subset $\mathcal{F}_s$ to define the state space (see Figure 3). Flags are first scored and selected conditioned on goal related criteria: (i) modeling goal related states; (ii) relations; and (iii) execution signs of actions. However, this approach may overlook Flags required for modeling intermediate but not goal-related actions. To address this, we perform scoring again with criteria strictly based on (i) action effects; and (ii) action preconditions and collect the union of candidate Flags in both rounds. This step recovers the Flags necessary for modeling intermediate actions, ensuring that all necessary state transitions are modeled within the generated logic program. A template of pre-built flags may be found in Appendix J.

3.5. Ground Rules Generation

With the $\mathcal{BC}+$ signature established, we synthesize the Ground Rules governing cause and effect. (see Figure 4). As preparation, the scope of influence for each Action is identified in form of grouping. An LLM isolates the subset of Flags $\mathcal{F}(a) \subseteq \mathcal{F}_s$ that are causally affected by a for

Table 1. Performance comparison and error analysis in VirtualHome. Noteably, Decover outperforms LLM-only baselines by up to ↑34.2%. in (Task SR), while reducing Missing Step errors by up to ↓26.9%.

Model	Overall Performance (↑)		Grammar Error (↓)		Runtime Error (↓)	
	Task SR	Exec SR	Parsing	Hallucination	Wrong Order	Missing Step
GPT-4o	72.8	80.3	0.0	0.7	0.3	18.0
DecoVer (*GPT-4o*)	82.9 (↑10.1)	89.9 (↑9.6)	5.7	0.0	0.0	4.4 (↓13.6)
GPT-5.1	63.3	66.6	0.0	28.5	0.0	4.3
DecoVer (*GPT-5.1*)	86.5 (↑23.2)	91.9 (↑25.3)	5.1	0.0	0.0	3.0 (↓1.3)
Claude-Opus-4.5	57.4	62.0	0.0	22.0	0.3	15.7
DecoVer (*Claude-Opus-4.5*)	91.6 (↑34.2)	94.0 (↑32.0)	3.0	0.0	0.0	3.0 (↓12.7)
Claude-Opus-4.5-thinking	72.8	77.0	0.3	5.2	0.0	14.8
DecoVer (*Claude-Opus-4.5-thinking*)	89.6 (↑16.8)	94.3 (↑17.3)	3.7	0.0	0.0	2.0 (↓12.7)
Gemini-3-Pro-Preview	76.7	84.3	3.0	0.0	0.0	12.1
DecoVer (*Gemini-3-Pro-Preview*)	89.6 (↑12.9)	92.3 (↑8.0)	4.7	0.0	0.3	2.7 (↓9.4)
Gemini-3-Pro-Preview-thinking	78.7	86.9	1.3	0.0	0.0	10.9
DecoVer (*Gemini-3-Pro-Preview-thinking*)	89.6 (↑10.9)	92.6 (↑5.7)	5.7	0.0	0.0	1.7 (↓9.2)
Qwen-Max-Latest	65.9	70.2	0.0	0.3	1.3	28.2
DecoVer (*Qwen-Max-Latest*)	88.3 (↑22.4)	94.0 (↑23.8)	4.7	0.0	0.0	1.3 (↓26.9)

all $a \in A_s$. Then for each pair (a, f) where $f \in \mathcal{F}(a)$, the model formulates the transition logic a **causes** $f = v$, explicitly determining the post-action value v.

Verification. Generated rules undergo a multi-round revision to ensure maximal correctness. A separate verifier LLM checks the rules for format and logical errors and if found, ground rules will be regenerated with the error feedback as extra information. This is repeated until all rules are valid or the revision round limit is reached.

3.6. Restrictions Generation

Finally, we generate restrictions $\mathcal{R}_r$ which prohibits invalid transitions (see Figure 5). Formally, a restriction is defined as **nonexecutable** $a(o)$ **if** ψ, where ψ is the prohibiting condition. To ensure restriction coverage, we decompose the acquisition of restrictions into two categories:

Target Object Restrictions. These restrictions enforce constraints derived from the target Object's properties and states. Let $\mathcal{F}(o)$ be the set of Flags modeling the target Object o. The LLM constructs restrictions where the condition ψ_{local} is a formula over $\mathcal{F}(o)$:

$$\text{nonexecutable } a(o) \text{ if } \psi_{local}(o)$$

Environment Object Restrictions. These restrictions enforce constraints imposed by the surrounding environment. Consider the set of Objects excluding o, The LLM constructs restrictions where the condition ψ_{global} involves spatial relations or states of external entities o':

$$\text{nonexecutable } a(o) \text{ if } \exists o' : \psi_{global}(o, o')$$

The primary objective of this decomposition is to suppress the tendency for LLMs to overlook complex environmental restrictions.

3.7. Theoretical Insight

We provide a formalization of the mechanism through which DecoVer's decomposition strategy mitigates the combinatorial explosion inherent in embodied task planning. Our approach achieves a heuristic reduction of the LLM's hypothesis space, thereby enhancing reasoning precision and effectively minimizing potential hallucinations.

Reduction of Reasoning Scope. Let $\mathcal{O}_{env}$ be the set of all objects in the environment. Without decomposition, synthesizing causal laws requires the LLM to implicitly navigate a candidate space of $\mathcal{S} = |\mathcal{A}| \cdot |\mathcal{F}|$, which scales quadratically with environment size ($S_{full} \propto |\mathcal{O}_{env}|^2$). DecoVer addresses this by projecting the open-world state onto a task-relevant subset $\mathcal{O}_s$ (where $|\mathcal{O}_s| \ll |\mathcal{O}_{env}|$). This foundational filter restricts the effective reasoning scope to $S_{decover} = |\mathcal{A}_s| \cdot |\mathcal{F}_s|$. By focusing the model's attention on this minimal manifold, DecoVer systematically filters open-world noise and reduces the likelihood of hallucinations during ground rule and restriction synthesis.

Structural Refinement via Verification. The Action Verification stage computes a bounded partial closure over the action set $\mathcal{A}_s$. Iteratively adding prerequisite actions a_{req} whose effects satisfy the preconditions of currently selected actions ($\text{Pre}(\mathcal{A}_s) \cap \text{Eff}(a_{req}) \neq \emptyset$). This is a monotonic operation that guarantees termination. In practice, DecoVer implements this as a single-pass verification to recover essential missing actions while preventing the over-expansion of the action space, thereby maintaining a compact transition system. The verification of ground rules acts as a strict semantic filter for the rule set $\mathcal{R}_c$. A generated rule r is pruned if it violates either of the following formal constraints:

- **Static Inconsistency**: $\text{Eff}(r) \cup \mathcal{K} \models \bot$, where $\mathcal{K}$ is the set of invariant static constraints. This prunes rules that would create physically impossible states.

Table 2. Performance comparison and error analysis on **logically extended** test cases in VirtualHome. Notably, DecoVer outperforms LLM-only baselines by up to ↑60.9% in Task Success Rate (Task SR), while reducing Missing Step errors by up to ↓21.7%.

Model	Overall Performance (↑)		Grammar Error (↓)		Runtime Error (↓)	
	Task SR	Exec SR	Parsing	Hallucination	Wrong Order	Missing Step
GPT-4o	34.8	26.1	0.0	4.3	21.7	39.1
DecoVer (*GPT-4o*)	39.1 (↑4.3)	43.5 (↑17.4)	26.1	0.0	0.0	26.1 (↓13.0)
GPT-5.1	13.0	13.0	0.0	65.2	4.3	17.4
DecoVer (*GPT-5.1*)	73.9 (↑60.9)	78.3 (↑65.3)	8.7	0.0	0.0	4.3 (↓13.1)
Claude-Opus-4.5	47.8	43.5	0.0	21.7	13.0	21.7
DecoVer (*Claude-Opus-4.5*)	87.0 (↑39.2)	91.3 (↑47.8)	4.3	0.0	4.3	0.0 (↓21.7)
Claude-Opus-4.5-thinking	47.8	47.8	0.0	34.8	4.3	13.0
DecoVer (*Claude-Opus-4.5-thinking*)	87.0 (↑39.2)	87.0 (↑39.2)	8.7	0.0	0.0	4.3 (↓8.7)
Gemini-3-Pro-Preview	78.3	78.3	8.7	0.0	13.0	0.0
DecoVer (*Gemini-3-Pro-Preview*)	87.0 (↑8.7)	87.0 (↑8.7)	8.7	0.0	0.0	4.3 (↑4.3)
Gemini-3-Pro-Preview-Thinking	73.9	73.9	0.0	0.0	8.7	17.4
DecoVer (*Gemini-3-Pro-Preview-Thinking*)	82.6 (↑8.7)	82.6 (↑8.7)	8.7	0.0	0.0	8.7 (↓8.7)
Qwen-Max-Latest	39.1	39.1	0.0	0.0	30.4	26.1
DecoVer (*Qwen-Max-Latest*)	60.9 (↑21.8)	60.9 (↑21.8)	21.7	0.0	0.0	17.4 (↓8.7)

- **Causal Misalignment**: $\mathrm{Var}(r) \notin \mathcal{M}(\mathrm{Act}(r))$, where $\mathcal{M}(a)$ denotes the mechanism scope of action a. This ensures the LLM's causal assertions remain within the legitimate functional boundaries of the specific action.

By enforcing verification, DecoVer ensures that the synthesized $\mathcal{BC}+$ program is not only logically self-consistent but also strictly grounded in the physical dynamics of the environment.

4. Experiments

4.1. Environmental Setup

Datasets. To comprehensively assess the performance of the proposed DecoVer on both embodied and traditional task planning, following (Li et al., 2024) and (Ishay & Lee, 2025), we evaluate the framework using the VirtualHome dataset (Puig et al., 2018) and the classical MCP (McCarthy, 1998) and Tower of Hanoi datasets, respectively. VirtualHome comprises 298 real-world household activities, such as "turn on light" and "cook some food", covering diverse task scenarios. To evaluate in complex settings, we augment VirtualHome with 23 logically extended cases featuring hard-to-identify preconditions and 20 temporally extended cases with long task horizons. The MCP dataset consists of two basic instances (Std1, Std2), five variations with special entities (Ent1–Ent5), and three with an expanded action space (Act1–Act3). In parallel, the Towers of Hanoi dataset covers 3-, 5-, and 7-disk instances, alongside corresponding variations initialized with arbitrarily scattered disks (denoted as 3/5/7 and 3/5/7_var, respectively). Please refer to Appendix E for further details.

Evaluation Metrics. As in (Li et al., 2024), we evaluate on VirtualHome and its extended cases using three primary metric categories. Specifically, Overall Performance includes Task Success Rate (Task SR) to measure goal satisfaction, and Execution Success Rate (Exec SR) to indicate error-free execution. Grammar Error accounts for format violations (Parsing), or unmentioned entities (Hallucination). Finally, Runtime Error encompasses Wrong Order for invalid temporal sequences, Missing Step for precondition violations.We further analyze Goal Satisfaction Rates across State (S), Relation (R), Action (A), and Total categories. In contrast, for MCP and Towers of Hanoi, we measure performance via Success Count (SC), defined as the number of solved instances.

4.2. Implementation Details

Baselines. To evaluate the effectiveness of DecoVer, we selected representative models spanning four families: GPT-4o and GPT-5.1 from GPT (OpenAI, 2024; Singh et al., 2025); Claude-Opus-4.5 and Claude-Opus-4.5-thinking from Claude (Anthropic, 2025); Gemini-3-Pro-Preview and Gemini-3-Pro-Preview-thinking from Gemini (DeepMind, 2025); and Qwen-Max-Latest (Yang et al., 2025) from Qwen.

Unless otherwise stated, we use 3 scoring rounds for object selecting, action selecting (increasing to 4 on retry), and Flag selecting. Ground rule verification allows up to 5 rounds. These hyperparameters were determined via empirical testing on a small validation set: reducing rounds below 3 caused a substantial performance degradation, while the current settings optimally balance accuracy and computational efficiency. All experiments are conducted on a Ubuntu 22.04, equipped with 32 vCPUs on an Intel(R) Xeon(R) Platinum 8352V CPU @ 2.10GHz.

4.3. Main Results

Results on VirtualHome. As shown in Tables 1 and 4, DecoVer significantly improves both overall performance and goal satisfaction rates across all baselines, effectively

Table 3. Performance comparison and error analysis on **temporally extended** test cases in VirtualHome. Notably, DecoVer outperforms LLM-only baselines by up to ↑45.0% in Task Success Rate (Task SR), while reducing Missing Step errors by up to ↓20.0%.

Model	Overall Performance (↑)		Grammar Error (↓)		Runtime Error (↓)	
	Task SR	Exec SR	Parsing	Hallucination	Wrong Order	Missing Step
GPT-4o	10.0	20.0	0.0	10.0	30.0	30.0
DecoVer (*GPT-4o*)	30.0 (↑20.0)	40.0 (↑20.0)	40.0	0.0	0.0	10.0 (↓20.0)
GPT-5.1	0.0	0.0	0.0	70.0	10.0	10.0
DecoVer (*GPT-5.1*)	30.0 (↑30.0)	30.0 (↑30.0)	35.0	0.0	0.0	10.0 (↓0.0)
Claude-Opus-4.5	50.0	50.0	0.0	45.0	0.0	5.0
DecoVer (*Claude-Opus-4.5*)	75.0 (↑25.0)	80.0 (↑30.0)	15.0	0.0	5.0	0.0 (↓5.0)
Claude-Opus-4.5-thinking	45.0	45.0	0.0	50.0	0.0	5.0
DecoVer (*Claude-Opus-4.5-thinking*)	90.0 (↑45.0)	90.0 (↑45.0)	10.0	0.0	0.0	0.0 (↓5.0)
Gemini-3-Pro-Preview	85.0	85.0	0.0	0.0	5.0	10.0
DecoVer (*Gemini-3-Pro-Preview*)	95.0 (↑10.0)	95.0 (↑10.0)	5.0	0.0	0.0	0.0 (↓10.0)
Gemini-3-Pro-Preview-thinking	85.0	85.0	0.0	0.0	0.0	10.0
DecoVer (*Gemini-3-Pro-Preview-thinking*)	90.0 (↑5.0)	95.0 (↑10.0)	5.0	0.0	0.0	0.0 (↓10.0)
Qwen-Max-Latest	30.0	35.0	0.0	15.0	15.0	15.0
DecoVer (*Qwen-Max-Latest*)	70.0 (↑40.0)	75.0 (↑40.0)	10.0	0.0	0.0	10.0 (↓5.0)

Table 4. Goal satisfaction comparison on VirtualHome. Notably, DecoVer outperforms LLM-only baselines by up to ↑40.0% in Total Goals.

Model	S	R	A	Total (↑)	Δ
GPT-4o	92.8	78.9	53.4	79.0	–
DecoVer (*GPT-4o*)	89.9	82.5	80.3	85.4	↑6.4
GPT-5.1	70.9	66.7	41.2	62.4	–
DecoVer (*GPT-5.1*)	96.8	83.6	86.4	90.3	↑27.9
Claude-Opus-4.5	77.0	46.7	19.6	54.0	–
DecoVer (*Claude-Opus-4.5*)	97.5	92.1	89.8	94.0	↑40.0
Claude-Opus-4.5-thinking	92.1	66.1	48.0	73.6	–
DecoVer (*Claude-Opus-4.5-thinking*)	98.6	90.4	83.7	92.5	↑18.9
Gemini-3-Pro-Preview	91.0	78.3	59.5	79.5	–
DecoVer (*Gemini-3-Pro-Preview*)	97.1	88.1	87.1	92.0	↑12.5
Gemini-3-Pro-Preview-thinking	96.8	78.3	60.8	82.5	–
DecoVer (*Gemini-3-Pro-Preview-thinking*)	93.5	87.0	91.2	91.0	↑8.5
Qwen-Max-Latest	78.8	79.4	50.7	72.1	–
DecoVer (*Qwen-Max-Latest*)	94.2	88.7	82.3	89.7	↑17.6

Table 5. Goal satisfaction comparison across different models on **logically extended** test cases in VirtualHome. Notably, DecoVer outperforms LLM-only baselines by up to ↑65.8% in Total Goals.

Model	S	R	A	Total (↑)	Δ
GPT-4o	53.8	36.4	42.9	45.6	–
DecoVer (*GPT-4o*)	59.0	39.4	57.1	50.6	↑5.0
GPT-5.1	28.2	0.0	0.0	13.9	–
DecoVer (*GPT-5.1*)	74.4	84.8	85.7	79.7	↑65.8
Claude-Opus-4.5	56.4	54.5	42.9	54.4	–
DecoVer (*Claude-Opus-4.5*)	89.7	93.9	100.0	92.4	↑38.0
Claude-Opus-4.5-thinking	71.8	24.2	71.4	51.9	–
DecoVer (*Claude-Opus-4.5-thinking*)	84.6	97.0	71.4	88.6	↑36.7
Gemini-3-Pro-Preview	82.1	69.7	100.0	78.5	–
DecoVer (*Gemini-3-Pro-Preview*)	82.1	93.9	100.0	88.6	↑10.1
Gemini-3-Pro-Preview-thinking	87.2	87.9	57.1	84.8	–
DecoVer (*Gemini-3-Pro-Preview-thinking*)	71.8	90.9	85.7	81.0	↓3.8
Qwen-Max-Latest	61.5	48.5	71.4	57.0	–
DecoVer (*Qwen-Max-Latest*)	53.8	63.6	100.0	62.0	↑5.1

mitigating the persistent sequential omission issues (i.e., 'Missing Step') common in LLM-based planners. Specifically, even against the powerful Gemini-3-Pro-Preview, DecoVer achieves a substantial 12.9% gain in Task SR. Notably, compared to Claude-Opus-4.5, it boosts the Total Goal Satisfaction Rate by 40%, demonstrating its robust capability in embodied task planning.

Logical or Temporal Extensions. We further evaluate the DecoVer framework on custom-designed logically and temporally extended test cases. Based on Tables 2 to 7, a clear distinction in model behavior emerges under these complex scenarios. Unlike their performance on standard VirtualHome, most vanilla LLM baselines experience significant performance degradation, particularly in temporally extended cases. This highlights that probabilistic semantic reasoning struggles to scale within strict physical environments. In contrast, DecoVer demonstrates consistent robustness. By offloading long-horizon state tracking to the $\mathcal{BC}+$ solver, DecoVer effectively prevents the compounding errors typical of vanilla LLMs. Notably, while advanced models like GPT-5.1 struggle independently on these custom extensions, they achieve substantial gains up to 60.9% when integrated into the DecoVer framework. This validates that our Decompose-and-Verify mechanism is essential for complex embodied planning. Full tables of tables 1-3 may be found in appendix B. Further analysis and corresponding cases are detailed in Appendices A,C and D.

Results on MCP and Tower of Hanoi. Results on traditional planning tasks (see Tables 6 and 8) reveal the following insights: 1) DecoVer significantly empowers the Claude family, enabling it to achieve a near-maximum Success Count. 2) The Gemini family exhibits robust baseline performance and, when combined with DecoVer, attains the maximum Success Count across all tasks. 3) Conversely, the GPT and Qwen families demonstrate weak baseline reasoning on these logically constrained tasks. This deficiency bottlenecks the efficacy of DecoVer, resulting in less pronounced improvements compared to the other families.

Table 6. Results on the MCP dataset. A green checkmark (✓) indicates a correct solution, while a red cross (✗) indicates any type of failure. The rightmost column displays the Success Count (SC), representing the total number of correctly solved instances.

Model	Std1	Std2	Ent1	Ent2	Ent3	Ent4	Ent5	Act1	Act2	Act3	SC
GPT-4o	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	0
DecoVer (*GPT-4o*)	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	1
GPT-5.1	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	1
DecoVer (*GPT-5.1*)	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	2
Claude-Opus-4.5	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	2
DecoVer (*Claude-Opus-4.5*)	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	9
Claude-Opus-4.5-thinking	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	8
DecoVer (*Claude-Opus-4.5-thinking*)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	10
Gemini-3-Pro-Preview	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	8
DecoVer (*Gemini-3-Pro-Preview*)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	10
Gemini-3-Pro-Preview-thinking	✓	✓	✗	✓	✓	✓	✓	✓	✗	✗	7
DecoVer (*Gemini-3-Pro-Preview-thinking*)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	10
Qwen-Max-Latest	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	0
DecoVer (*Qwen-Max-Latest*)	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	1

Table 7. Goal satisfaction comparison on **temporally extended** test cases in VirtualHome. Notably, DecoVer outperforms LLM-only baselines by up to ↑39.9% in Total Goals.

Model	S	R	A	Total (↑)	Δ
GPT-4o	60.0	33.3	33.3	42.9	–
DecoVer (*GPT-4o*)	43.3	40.2	33.3	41.1	↓1.8
GPT-5.1	15.0	2.9	16.7	7.7	–
DecoVer (*GPT-5.1*)	33.3	36.3	50.0	35.7	↑28.0
Claude-Opus-4.5	66.7	44.1	33.3	51.8	–
DecoVer (*Claude-Opus-4.5*)	83.3	82.4	66.7	82.1	↑30.3
Claude-Opus-4.5-thinking	60.0	44.1	33.3	49.4	–
DecoVer (*Claude-Opus-4.5-thinking*)	85.0	91.2	100.0	89.3	↑39.9
Gemini-3-Pro-Preview	98.3	95.1	66.7	95.2	–
DecoVer (*Gemini-3-Pro-Preview*)	91.7	97.1	100.0	95.2	↑0.0
Gemini-3-Pro-Preview-thinking	98.3	97.1	66.7	96.4	–
DecoVer (*Gemini-3-Pro-Preview-thinking*)	91.7	95.1	100.0	94.0	↓2.4
Qwen-Max-Latest	48.3	54.9	0.0	50.6	–
DecoVer (*Qwen-Max-Latest*)	80.0	79.4	100.0	80.4	↑29.8

Table 8. Results on the Hanoi dataset. A green checkmark (✓) indicates a correct solution, while a red cross (✗) denotes any type of failure. The rightmost column displays the Success Count (SC), representing the total number of correctly solved instances.

Model	3	3_var	5	5_var	7	7_var	SC
GPT-4o	✓	✗	✗	✗	✗	✗	1
DecoVer (*GPT-4o*)	✓	✓	✗	✗	✗	✗	2
GPT-5.1	✓	✗	✓	✗	✓	✗	3
DecoVer (*GPT-5.1*)	✓	✓	✓	✓	✓	✓	6
Claude-Opus-4.5	✓	✗	✓	✗	✓	✗	3
DecoVer (*Claude-Opus-4.5*)	✓	✓	✓	✓	✓	✓	6
Claude-Opus-4.5-thinking	✓	✓	✓	✓	✓	✗	5
DecoVer (*Claude-Opus-4.5-thinking*)	✓	✓	✓	✓	✓	✓	6
Gemini-3-Pro-Preview	✓	✓	✓	✓	✓	✓	6
DecoVer (*Gemini-3-Pro-Preview*)	✓	✓	✓	✓	✓	✓	6
Gemini-3-Pro-thinking	✓	✓	✓	✓	✓	✓	6
DecoVer (*Gemini-3-Pro-thinking*)	✓	✓	✓	✓	✓	✓	6
Qwen-Max-Latest	✗	✗	✗	✗	✗	✗	0
DecoVer (*Qwen-Max-Latest*)	✗	✓	✓	✓	✗	✗	3

Details of these experiments can be found in Appendix E.

4.4. Error Analysis

In Virtualhome, the most pervasive failure mode for vanilla LLMs is the **Missing Step** error, which primarily manifests in two forms: Attempting to interact with an object without navigating to its location, or overlooking essential action prerequisites. Also, we observe that Claude-Opus-4.5 and its thinking variant along with GPT-5.1 append numeric suffixes to actions to force JSON format validity, triggering massive hallucination errors. While DecoVer successfully mitigates these failures above, it is not immune to failure. Most notably, the generation of overly strict restrictions can inadvertently over-constrain the environment, pruning valid action trajectories and causing the solver to yield a **No Solution** result. This issue is compounded by a structural vulnerability: because the restriction generation phase currently operates without verification, the synthesized restrictions remain susceptible to syntax errors. Beyond these rule-level challenges, the framework is also sensitive to upstream dependencies; any omission of critical objects, actions, or flags from the initial candidate sets can cascade through the pipeline, ultimately rendering the final program unexecutable. On standard Virtualhome cases, **98.3%** of the generated $\mathcal{BC}+$ programs possessed valid syntax. Of these syntactically correct programs, **97.0%** were able to produce a solution. Further experimental analysis and detailed error cases are provided in Appendix B.

4.5. DecoVer vs. LLM+AL

Given that LLM+AL is the sole neuro-symbolic approach integrating LLMs with $\mathcal{BC}+$, we conducted a comparative study on 100 randomly sampled VirtualHome instances. We benchmarked DecoVer against LLM+AL and vanilla LLM baselines (using Gemini-3-Pro-Preview and GPT-4o as backbones) in terms of Task SR and Missing Step. Figure 6 demonstrates that DecoVer consistently outperforms all competitors. Notably, LLM+AL trails behind the vanilla LLMs on both backbones. This performance gap primarily stems from two critical bottlenecks in the LLM+AL

Table 9. Ablation study on 11 challenging instances in VirtualHome. The decomposition mechanism of DecoVer consists of five stages (S1-S5). Δ indicates the performance degradation compared to the complete DecoVer framework.

Configuration	Claude-Opus-4.5		Gemini-3-Pro-Preview		Qwen-Max-Latest	
	Task SR(%)	Δ	Task SR(%)	Δ	Task SR(%)	Δ
DecoVer	**81.8**	-	**90.9**	-	**72.7**	-
Verification Mechanism						
w/o Action Verification	54.5	-27.3	90.9	0.0	36.4	-36.3
w/o Ground Rules Verification	72.7	-9.1	81.8	-9.1	54.5	-18.2
Decomposition Mechanism						
w/o S1	81.8	0.0	81.8	-9.1	63.6	-9.1
All Objects in S2	54.5	-27.3	90.9	0.0	45.5	-27.2
All Objects and Actions in S3	36.4	-45.4	45.5	-45.4	45.5	-27.2
w/o Flag Grouping in S4	36.4	-45.4	45.5	-45.4	27.3	-45.4
w/o Decomposition in S5	63.6	-18.2	63.6	-27.3	63.6	-9.1
Scoring Rounds						
1 Round	63.6	-18.2	81.8	-9.1	72.7	0.0
5 Rounds	72.7	-9.1	81.8	-9.1	72.7	0.0

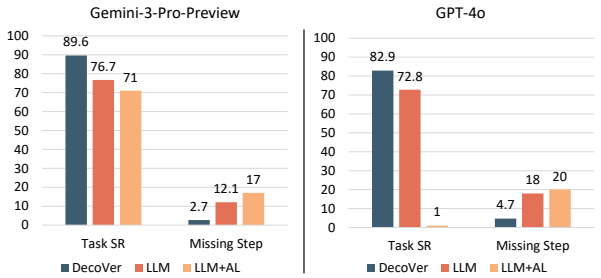

Figure 6. Comparison of Task SR and Missing Step across Gemini-3-Pro-Preview and GPT-4o: DecoVer consistently surpasses LLM+AL, whereas LLM+AL falls behind the vanilla LLM.

paradigm. First, LLM+AL relies on solver errors for syntax refinement and sample query results for logical refinement, yet both sources offer insufficient guidance. The solver's error messages can be too cryptic to guide effective syntactic corrections. Simultaneously, mapping a sample query's erroneous execution result back to its root cause within the entire $\mathcal{BC}+$ program creates an overwhelmingly large search space for logical refinement. Conversely, DecoVer isolates refinement to granular modules. By providing localized and highly precise context, our approach enables the LLM to leverage its reasoning capabilities to correct specific errors more effectively. Second, the initial knowledge extraction phase of LLM+AL which builds the foundational blueprint of the $\mathcal{BC}+$ program is finished in a single pass without verification. Consequently, flawed knowledge inevitably produces programs that are physically ungrounded despite being logically self-consistent. In contrast, DecoVer applies verification at critical steps, which ensures that intermediate representations are rigorously aligned with physical constraints before execution. Further details of this experiment may be found in Appendix G.

4.6. Ablation Studies

For further details of our Ablation Studies, please refer to Appendix F. **Effect of Decomposition.** Given that the generation of Ground Rules and Restrictions is contingent upon the first three steps (Object, Action, and Flag Selecting), we focus our ablation study on these foundational modules. We curated 11 challenging, error-prone instances from VirtualHome to evaluate their impact. As shown in Table 9, ablating any of these steps causes a sharp decline in performance across all three LLMs, underscoring the necessity and effectiveness of the proposed decomposition strategy.

Effect of Verification. Complementing the decomposition mechanism, the verification mechanism plays a critical role. In this work, we focus verification efforts on the high-risk stages of Action Selecting and Ground Rules, with further extensions planned for future research. Ablation results (Table 9) indicate that sensitivities vary across models. Specifically, Claude-Opus-4.5 and Qwen-Max-Latest favor Action Verification, whereas Gemini-3-Pro-Preview benefits more from Ground Rules Verification. Nevertheless, the verification mechanism universally improves the Task SR.

Effect of Scoring Rounds. Scoring rounds necessitate a balance between performance and cost. We compared our default setting (3-5 rounds) against uniform 1- and 5-round baselines. Table 9 shows that 1 round is insufficient, causing performance declines. Meanwhile, extending to 5 rounds provides no superior results compared to the default but introduces unnecessary latency, lowering overall efficiency.

5. Conclusion

In this work, we propose DecoVer, a Decompose-and-Verify neuro-symbolic framework that enhances LLM-based embodied task planning by grounding natural language tasks into executable $\mathcal{BC}+$ programs. Extensive experiments show that through structured decomposition and iterative verification, DecoVer generates high-fidelity formalisms, effectively mitigating the causal inconsistency and sequential omission errors inherent in direct end-to-end methods.

Despite these advancements, limitations persist. First, verification currently targets only critical stages; future work should explore efficient strategies covering the entire pipeline. Second, our $\mathcal{BC}+$ framework is confined to LLMs; extending it to MLLMs to integrate multimodal data remains a key research direction.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (No. U25A20442, and 62427808).

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal

consequences of our work, none which we feel must be specifically highlighted here.

References

Ahn, S., Choi, W., Lee, J., Park, J., and Woo, H. Towards reliable code-as-policies: A neuro-symbolic framework for embodied task planning. *Advances in Neural Information Processing Systems*, 38:75428–75459, 2026.

Anthropic. Model card addendum: Claude 4.5 opus and sonnet, 2025.

Babb, J. and Lee, J. Action language bc+: preliminary report. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

Babb, J. and Lee, J. Action language bc+. *Journal of Logic and Computation*, 30(4):899–922, 2020.

Brohan, A., Chebotar, Y., Finn, C., Hausman, K., Herzog, A., Ho, D., Ibarz, J., Irpan, A., Jang, E., Julian, R., et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on robot learning*, pp. 287–318. PMLR, 2023.

Chen, Y., Arkin, J., Dawson, C., Zhang, Y., Roy, N., and Fan, C. Autotamp: Autoregressive task and motion planning with llms as translators and checkers. In *2024 IEEE International conference on robotics and automation (ICRA)*, pp. 6695–6702. IEEE, 2024.

Choi, W., Park, J., Ahn, S., Lee, D., and Woo, H. Nesyc: A neuro-symbolic continual learner for complex embodied tasks in open domains. In *The Thirteenth International Conference on Learning Representations*, 2025.

DeepMind, G. Gemini 3: The frontier of vision and reasoning ai. Google DeepMind Blog, 2025.

Fang, M., Deng, S., Zhang, Y., Shi, Z., Chen, L., Pechenizkiy, M., and Wang, J. Large language models are neurosymbolic reasoners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 17985–17993, 2024.

Gelfond, M. and Lifschitz, V. Representing action and change by logic programs. *The Journal of Logic Programming*, 17(2-4):301–321, 1993.

Gelfond, M. and Lifschitz, V. *Action languages*. Linköping University Electronic Press, 1998.

Gestrin, E., Kuhlmann, M., and Seipp, J. Towards robust LLM-driven planning from minimal text descriptions. In *ICAPS 2024 Workshop on Human-Aware Explainable Planning*, 2024.

Giunchiglia, E. and Lifschitz, V. An action language based on causal explanation: Preliminary report. In *AAAI*, pp. 623–630, 1998.

Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., and Turner, H. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1-2):49–104, 2004.

Hao, S., Gu, Y., Ma, H., Hong, J., Wang, Z., Wang, D., and Hu, Z. Reasoning with language model is planning with world model. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 8154–8173, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.507.

Hao, Y., Zhang, Y., and Fan, C. Planning anything with rigor: General-purpose zero-shot planning with llm-based formalized programming. In *International Conference on Learning Representations*, volume 2025, pp. 64335–64391, 2025.

Ishay, A. and Lee, J. LLM+AL: Bridging large language models and action languages for complex reasoning about actions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 24212–24220, 2025.

Lee, J., Lifschitz, V., and Yang, F. Action language bc: Preliminary report. In *IJCAI*, pp. 983–989, 2013.

Li, B., Silver, T., Scherer, S., and Gray, A. Bilevel learning for bilevel planning. *arXiv preprint arXiv:2502.08697*, 2025a.

Li, M., Zhao, S., Wang, Q., Wang, K., Zhou, Y., Srivastava, S., Gokmen, C., Lee, T., Li, E. L., Zhang, R., et al. Embodied agent interface: Benchmarking llms for embodied decision making. *Advances in Neural Information Processing Systems*, 37:100428–100534, 2024.

Li, X., He, X., Zhang, L., Wu, M., Li, X., and Liu, Y. A comprehensive survey on world models for embodied ai, 2025b.

Li, Z., Lu, F., Fu, T., and Tian, G. From demand to grounded plan: Task customization and planning for service robots with deep learning and llms. *IEEE Robotics and Automation Letters*, 2025c.

Lin, X., Wu, Y., Yang, H., Huang, Y., Zhang, Y., Ji, J., and Zhang, Y. Improving efficiency of answer set planning with rough solutions from large language models for robotic task planning. In Kwok, J. (ed.), *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pp. 4570–4578. International Joint Conferences on Artificial Intelligence Organization, 8 2025. doi: 10.24963/ijcai.2025/509. Main Track.

Ling, H., Parashar, S., Khurana, S., Olson, B., Basu, A., Sinha, G., Tu, Z., Caverlee, J., and Ji, S. Complex llm planning via automated heuristics discovery. *arXiv preprint arXiv:2502.19295*, 2025a.

Ling, H., Parashar, S., Khurana, S., Olson, B., Basu, A., Sinha, G., Tu, Z., Caverlee, J., and Ji, S. Complex llm planning via automated heuristics discovery, 2025b.

Liu, B., Jiang, Y., Zhang, X., Liu, Q., Zhang, S., Biswas, J., and Stone, P. LLM+P: Empowering large language models with optimal planning proficiency. *arXiv e-prints*, pp. arXiv–2304, 2023.

Liu, X., Toroghi, A., Liang, J., Courtis, D., Li, R., Pesaranghader, A., Kim, J., Sadhu, T., Jeon, H., and Sanner, S. Natural language PDDL (NL-PDDL) for open-world goal-oriented commonsense regression planning in embodied AI. In *The Fourteenth International Conference on Learning Representations*, 2026.

McCarthy, J. Elaboration tolerance. In *Working Papers of the Fourth Symposium on Logical Formalizations of Commonsense Reasoning*, 1998.

OpenAI. Gpt-4o system card. *OpenAI*, 2024.

Puig, X., Ra, K., Boben, M., Li, J., Wang, T., Fidler, S., and Torralba, A. Virtualhome: Simulating household activities via programs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8494–8502, 2018.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Singh, A., Fry, A., Perelman, A., Tart, A., Ganesh, A., El-Kishky, A., McLaughlin, A., Low, A., Ostrow, A., Ananthram, A., et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Valmeekam, K., Marquez, M., Sreedharan, S., and Kambhampati, S. On the planning abilities of large language models-a critical investigation. *Advances in Neural Information Processing Systems*, 36:75993–76005, 2023.

Verma, P., La, N., Favier, A., Mishra, S., and Shah, J. A. Teaching llms to plan: Logical chain-of-thought instruction tuning for symbolic planning. *arXiv preprint arXiv:2509.13351*, 2025.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856.

Wang, S., Fei, Z., Cheng, Q., Zhang, S., Cai, P., Fu, J., and Qiu, X. World modeling makes a better planner: Dual preference optimization for embodied task planning. In *ACL*, pp. 21518–21537, 2025.

Wu, Z., Wang, Z., Xu, X., Lu, J., and Yan, H. Embodied task planning with large language models. *arXiv preprint arXiv:2307.01848*, 2023.

Yang, A. et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yang, Z., Ishay, A., and Lee, J. Coupling large language models with logic programming for robust and general reasoning from text. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 5186–5219, 2023.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 11809–11822. Curran Associates, Inc., 2023.

Zhou, Z., Song, J., Yao, K., Shu, Z., and Ma, L. Isr-llm: Iterative self-refined large language model for Long-Horizon sequential task planning. In *Proceedings-IEEE International Conference on Robotics and Automation*, pp. 2081–2088. Institute of Electrical and Electronics Engineers Inc., 2024.

Appendix Contents

A Anomalous Results Analysis . . . 12
B Analysis on VirtualHome . . . 13
B.1 LLM-Planner Errors . . . 13
B.2 DecoVer Errors . . . 15
C Analysis in Logically Extended Cases . . . 18
C.1 Cases . . . 18
C.2 Errors . . . 21
D Analysis in Temporally Extended Cases . . . 23
D.1 Cases . . . 23
D.2 Errors . . . 25
E Analysis in MCP and Hanoi . . . 27
F Analysis in Ablation Studies . . . 30
G DecoVer vs LLM+AL . . . 32
H Supplementary Experiments . . . 37
I Pipeline Details . . . 37
J Pipeline Utilities . . . 46

A. Abnormal Results In Extended Cases

In our evaluation, we observed a seemingly paradoxical phenomenon: while DecoVer achieves higher Task SR compared to baseline LLMs, it occasionally records slightly lower goal success rates in extended cases.

The core cause lies in the distinct failure modes of the two approaches:

- **DecoVer (Binary Failure):** As a neuro-symbolic system, DecoVer relies on a solver backend. If the generated $\mathcal{BC}+$ program contains a syntax error or a logical contradiction, no action sequence is produced, resulting in **zero** actions executed. Consequently, the goal satisfaction rates for that instance is 0%, punishing the aggregate metric heavily.
- **Baseline LLMs (Partial Failure):**When baseline LLMs fail, they often execute a significant portion of the plan correctly before the simulator reports a runtime error. This allows them to satisfy several intermediate goal conditions even though the overall task is marked as a failure.

We illustrate this utilizing the trace logs from the *Gemini-3-pro-preview-thinking* experiments on the temporally extended cases. We also confirm that other similar anomalies in **GPT-5.1** occured for the same reason.

Table 10. Comparative breakdown of failure penalties. Baseline LLMs inflate Goal Success Rates by accumulating partial credit on failed tasks, whereas DecoVer suffers binary penalties (e.g., Case 16).

Metric	**DecoVer**	**Baseline LLM**	**Impact Analysis**
Task Success Rate	**90.0%** (18/20)	85.0% (17/20)	DecoVer solves more full tasks.
Goal Success Rate	94.05%	**96.43%**	Baseline accumulates partial credit.
Failure Case Analysis			
Failed Instances	Case 16: Failure (0/8 Goals met) Case 20: Wrong Execution Order (9/11 Goals met)	Case 03: Affordance Err (2/5 Goals met) Case 13: Missing Step (4/5 Goals met) Case 14: Missing Step (5/7 Goals met)	Baseline collects **11/17** partial goals DecoVer collects **9/19** partial goals *(Heavily penalized by Case 16)*

As shown in Table 10, the Baseline LLM failed three tasks but successfully credited 11 intermediate goals across them. In contrast, DecoVer failed fewer tasks (only 2), but the failure on Case 16 disproportionately impacted the average goal metric.

B. Error Analysis On VirtualHome

Table 11. Full table of Table 1

Model	Overall Performance (↑)		Grammar Error (↓)			Runtime Error (↓)			
	Task SR	Exec SR	Parsing	Hallucination	Arg Num	Wrong Order	Missing Step	Affordance	Add. Step
GPT-4o	72.8	80.3	0.0	0.7	0.0	0.3	18.0	0.7	3.6
DecoVer (*GPT-4o*)	82.9 (↑10.1)	89.9 (↑9.6)	5.7	0.0	0.0	0.0	4.4 (↓13.6)	0.0	0.3
GPT-5.1	63.3	66.6	0.0	28.5	0.7	0.0	4.3	0.0	1.0
DecoVer (*GPT-5.1*)	86.5 (↑23.2)	91.9 (↑25.3)	5.1	0.0	0.0	0.0	3.0 (↓1.3)	0.0	0.3
Claude-Opus-4.5	57.4	62.0	0.0	22.0	0.0	0.3	15.7	0.0	2.3
DecoVer (*Claude-Opus-4.5*)	91.6 (↑34.2)	94.0 (↑32.0)	3.0	0.0	0.0	0.0	3.0 (↓12.7)	0.0	0.7
Claude-Opus-4.5-thinking	72.8	77.0	0.3	5.2	2.6	0.0	14.8	0.0	0.0
DecoVer (*Claude-Opus-4.5-thinking*)	89.6 (↑16.8)	94.3 (↑17.3)	3.7	0.0	0.0	0.0	2.0 (↓12.7)	0.0	0.0
Gemini-3-Pro-Preview	76.7	84.3	3.0	0.0	0.7	0.0	12.1	0.0	2.6
DecoVer (*Gemini-3-Pro-Preview*)	89.6 (↑12.9)	92.3 (↑8.0)	4.7	0.0	0.0	0.3	2.7 (↓9.4)	0.0	0.0
Gemini-3-Pro-Preview-thinking	78.7	86.9	1.3	0.0	1.3	0.0	10.9	0.0	2.6
DecoVer (*Gemini-3-Pro-Preview-thinking*)	89.6 (↑10.9)	92.6 (↑5.7)	5.7	0.0	0.0	0.0	1.7 (↓9.2)	0.0	0.0
Qwen-Max-Latest	65.9	70.2	0.0	0.3	0.0	1.3	28.2	0.0	3.3
DecoVer (*Qwen-Max-Latest*)	88.3 (↑22.4)	94.0 (↑23.8)	4.7	0.0	0.0	0.0	1.3 (↓26.9)	0.0	0.0

Table 12. Full Table of Table 2

Model	Overall Performance (↑)		Grammar Error (↓)			Runtime Error (↓)			
	Task SR	Exec SR	Parsing	Hallucination	Arg Num	Wrong Order	Missing Step	Affordance	Add. Step
GPT-4o	34.8	26.1	0.0	4.3	8.7	21.7	39.1	0.0	4.3
DecoVer (*GPT-4o*)	39.1 (↑4.3)	43.5 (↑17.4)	26.1	0.0	0.0	0.0	26.1 (↓13.0)	4.3	0.0
GPT-5.1	13.0	13.0	0.0	65.2	0.0	4.3	17.4	0.0	0.0
DecoVer (*GPT-5.1*)	73.9 (↑60.9)	78.3 (↑65.3)	8.7	0.0	0.0	0.0	4.3 (↓13.1)	8.7	4.3
Claude-Opus-4.5	47.8	43.5	0.0	21.7	0.0	13.0	21.7	0.0	0.0
DecoVer (*Claude-Opus-4.5*)	87.0 (↑39.2)	91.3 (↑47.8)	4.3	0.0	0.0	4.3	0.0 (↓21.7)	0.0	0.0
Claude-Opus-4.5-thinking	47.8	47.8	0.0	34.8	0.0	4.3	13.0	0.0	0.0
DecoVer (*Claude-Opus-4.5-thinking*)	87.0 (↑39.2)	87.0 (↑39.2)	8.7	0.0	0.0	0.0	4.3 (↓8.7)	0.0	0.0
Gemini-3-Pro-Preview	78.3	78.3	8.7	0.0	0.0	13.0	0.0	0.0	0.0
DecoVer (*Gemini-3-Pro-Preview*)	87.0 (↑8.7)	87.0 (↑8.7)	8.7	0.0	0.0	0.0	4.3 (↑4.3)	0.0	0.0
Gemini-3-Pro-Preview-Thinking	73.9	73.9	0.0	0.0	0.0	8.7	17.4	0.0	2.6
DecoVer (*Gemini-3-Pro-Preview-Thinking*)	82.6 (↑8.7)	82.6 (↑8.7)	8.7	0.0	0.0	0.0	8.7 (↓8.7)	0.0	0.0
Qwen-Max-Latest	39.1	39.1	0.0	0.0	0.0	30.4	26.1	4.3	0.0
DecoVer (*Qwen-Max-Latest*)	60.9 (↑21.8)	60.9 (↑21.8)	21.7	0.0	0.0	0.0	17.4 (↓8.7)	0.0	0.0

Table 13. Full Table of Table 3

Model	Overall Performance (↑)		Grammar Error (↓)			Runtime Error (↓)			
	Task SR	Exec SR	Parsing	Hallucination	Arg Num	Wrong Order	Missing Step	Affordance	Add. Step
GPT-4o	10.0	20.0	0.0	10.0	0.0	30.0	30.0	10.0	5.0
DecoVer (*GPT-4o*)	30.0 (↑20.0)	40.0 (↑20.0)	40.0	0.0	0.0	0.0	10.0 (↓20.0)	10.0	0.0
GPT-5.1	0.0	0.0	0.0	70.0	0.0	10.0	10.0	10.0	0.0
DecoVer (*GPT-5.1*)	30.0 (↑30.0)	30.0 (↑30.0)	35.0	0.0	0.0	0.0	10.0 (↓0.0)	25.0	0.0
Claude-Opus-4.5	50.0	50.0	0.0	45.0	0.0	0.0	5.0	0.0	0.0
DecoVer (*Claude-Opus-4.5*)	75.0 (↑25.0)	80.0 (↑30.0)	15.0	0.0	0.0	5.0	0.0 (↓5.0)	0.0	0.0
Claude-Opus-4.5-thinking	45.0	45.0	0.0	50.0	0.0	0.0	5.0	0.0	0.0
DecoVer (*Claude-Opus-4.5-thinking*)	90.0 (↑45.0)	90.0 (↑45.0)	10.0	0.0	0.0	0.0	0.0 (↓5.0)	0.0	0.0
Gemini-3-Pro-Preview	85.0	85.0	0.0	0.0	0.0	5.0	10.0	0.0	0.0
DecoVer (*Gemini-3-Pro-Preview*)	95.0 (↑10.0)	95.0 (↑10.0)	5.0	0.0	0.0	0.0	0.0 (↓10.0)	0.0	0.0
Gemini-3-Pro-Preview-thinking	85.0	85.0	0.0	0.0	0.0	0.0	10.0	5.0	0.0
DecoVer (*Gemini-3-Pro-Preview-thinking*)	90.0 (↑5.0)	95.0 (↑10.0)	5.0	0.0	0.0	0.0	0.0 (↓10.0)	0.0	0.0
Qwen-Max-Latest	30.0	35.0	0.0	15.0	0.0	15.0	15.0	20.0	5.0
DecoVer (*Qwen-Max-Latest*)	70.0 (↑40.0)	75.0 (↑40.0)	10.0	0.0	0.0	0.0	10.0 (↓5.0)	5.0	0.0

B.1. LLM-Planner Errors

In this section, we analyze the failure modes of the direct LLM-based planning baselines.

B.1.1. Hallucination Errors

A distinct anomaly was observed in the performance of high-capability closed-source models, specifically *GPT-5.1* and *Claude-4.5-Opus*. Despite their advanced reasoning, these models exhibited unexpectedly high "Hallucination Rates" (Table

1).

Analysis reveals this is driven by a misalignment between the models' strict adherence to JSON validity and the evaluation harness's flexible parsing strategy. While the benchmark prompt requests JSON-formatted output, the evaluation harness parses the response as a text stream rather than loading the JSON block directly.

However, certain models treat the instruction to output JSON as a strict constraint. Recognizing that standard JSON objects cannot contain duplicate keys, these models appending numerical suffixes or white spaces to ensure the output is a valid JSON object. This act causes the evaluation modules which matches action strings exactly against the ground truth objects and actions to flag these suffixed actions as hallucinations.

Over-Compliance with JSON Syntax

```
% The Model, enforcing valid JSON, "fixes" the keys:
"actions": [
    { "WALK":       ["light", 245] },

    % Suffix added to satisfy JSON uniqueness constraint
    { "WALK_1":     ["light", 1000] },

    { "WALK_2":     ["light", 411] }
]
% Result: "WALK_1" is considered as a non-existent action.
```

B.1.2. Missing Step Errors

The most pervasive failure mode for direct LLM planners is the **Missing Step** error. We categorize these failures into two distinct types: Spatial Neglect and Precondition Neglect.

Spatial Neglect. These errors occur when the model fails to track the agent's location relative to objects. This is particularly common when task horizons expand, and navigating to multiple intermediate destinations is required.

Spatial Neglect

```
"actions": [
    "[WALK] <plate> (1000)",
    "[GRAB] <plate> (1000)",
    "[GRAB] <dish_soap> (1001)",  <-- ERROR: Missing Step
    "[WASH] <plate> (1000)"
]


"actions": [
    "[WALK] <dining_room> (201)",
    "[WALK] <pot> (1000)",
    % Model assumes it can put the pot in the oven from the pot's location
    "[PUTIN] <pot> (1000) <oven> (295)", <-- ERROR: Missing Step
    "[WALK] <oven> (295)",
    "[SWITCHON] <oven> (295)"
]
```

Analysis: The first case requires walking to the dish_soap which is not close to the plate, while the second case requires walking to the oven. These intermediate actions were omitted.

Precondition Neglect. These errors reflect a lack of understanding regarding fine-grained physical constraints. For example, trying to drink from a cup without grabbing the cup, looking at something without turning towards it, and so on.

Precondition Neglect

```
"actions": [
    "[WALK] <nightstand> (100)",
    "[OPEN] <nightstand> (100)",
    "[GRAB] <novel> (1000)",
    "[LOOKAT] <novel> (1000)" <-- ERROR: Missing Step
]


"actions": [
    "[WALK] <dining_room> (201)",
    "[DRINK] <cup> (1000)" <-- ERROR: Missing Step
]
```

B.2. DecoVer Errors

While DecoVer significantly outperforms baselines, it is not immune to failure. When DecoVer fails, it typically returns an empty plan. We analyze the erroneous programs and classify DecoVer's runtime errors into three main types.

B.2.1. Type I: Over-Strict Constraints

In these cases, the LLM generates a constraint that is syntactically valid but logically too aggressive, inadvertently blocking necessary actions.

Over-Strict Constraints

```
:- sorts
    entity >> main_entity, static_entity; ...
:- objects
    void :: static_entity; ...
:- variables
    se, se1 :: static_entity; ...
:- constants
    grab(static_entity) :: exogenousAction;
    object_is_inside(static_entity) :: inertialFluent(static_entity);
    ...

% [CRITICAL LOGIC ERROR]
% The variable 'se1' binds to ANY static_entity, including 'void'.
% Since initial state is usually 'inside(obj)=void', this rule blocks grabbing EVERYTHING.
nonexecutable grab(se) if object_is_inside(se)=se1.

noconcurrency.
:- query
    0: ... object_is_inside(phone_1000)=void ...;
    maxstep: object_is_grabbed(phone_1000)=true.
```

Analysis: The model attempted to express "You cannot grab an object if it is inside another object." However, by failing to exclude the placeholder `void` (e.g., `se1 \= void`), the constraint universally bans grabbing any object that isn't inside a container, rendering the task unsolvable.

B.2.2. Type II: Syntax and Signature Mismatches

These errors occur when the model hallucinates language features or fails to align the restrictions with the declared signature. Although we provide detailed format instruction prompts, we cannot exterminate format errors entirely.

Syntax Error(Sleep)

```
:- constants
    sleep :: exogenousAction; % Defined as Arity 0 (Atomic)
    ...

% [SYNTAX ERROR]
% The constraint attempts to pass an argument 'se' to 'sleep',
% which violates the 0-arity definition.
nonexecutable sleep(se) if (character_is_lying=false).

noconcurrency.
```

Analysis: The model used the `SLEEP` action in a wrong format: `SLEEP` accepts zero parameters while the model passed one argument, which caused an execution failure.

Syntax Error (Undefined Flag)

```
:- constants
    switchon(static_entity) :: exogenousAction;
    object_is_switched_on(static_entity) :: inertialFluent(boolean);
    % [NOTE: 'object_is_plugged_in' is NOT defined here]

% [SIGNATURE ERROR]
% This rule relies on a fluent that does not exist in the signature.
nonexecutable switchon(se) if (object_is_plugged_in(se)=false).

noconcurrency.
```

Analysis: The model hallucinated a common-sense precondition ("must be plugged in") observed in few-shot examples, but failed to notice the corresponding flag was not selected.

B.2.3. Type III: Dependency Issues

These errors reveal problems within the selection stages. Such errors occur because of un-grouped flags, missing intermediate actions, and so on.

Missing Prerequisite Action

```
:- constants
    drink(static_entity) :: exogenousAction;
    character_facing :: inertialFluent(static_entity);
    % [NOTE: 'turnto' action is MISSING]

% [DEPENDENCY ERROR]
% This restriction demands the agent face the object.
nonexecutable drink(se) if (character_facing\=se).

% However, without 'turnto', 'character_facing' can never change
% from its initial state (void).
```

Analysis: The restriction "must face object to drink" is redundant. What's more, the Action Selection module did not consider `TURNTO` a necessary action for this task. Consequently, the precondition is unsatisfiable.

B.2.4. Type IV: Pipeline Integration Failures

Rarely, errors occur in the Python binding layer rather than the logic solver.

Hallucinated Flag

```
ERROR in 570_1:
'character_is_facing'
Traceback (most recent call last):
  File ".../run.py", line 133, in <module>
    program = generate_bc_program(...)
  File ".../build_bc.py", line 132, in generate_bc_program
    if flag_param_type_dict[flag]!="void":
KeyError: 'character_is_facing'
```

Analysis: The model hallucinated a flag name (`character_is_facing`) that did not exist in the system's predefined dictionary, causing a Python-level crash during the translation phase before the $\mathcal{BC}+$ program could even be generated.

B.2.5. Missing Step Errors

In these instances, the solver successfully finds a plan because the discovered constraints are too lax. The model fails to identify a critical precondition required, allowing the solver to generate a sequence that is logically valid within the $\mathcal{BC}+$ program but physically invalid in the environment.

Missing Step due to overlooked preconditions

```
:- constants
    read(static_entity) :: exogenousAction;
    object_is_grabbed(static_entity) :: inertialFluent(boolean);
    ...

% Generated constraints for 'read'
nonexecutable read(se) if (object_is_open(se)=false).
nonexecutable read(se) if (character_facing\=se).
nonexecutable read(se) if character_location \= object_location(se).

% [MISSING CONSTRAINT]
% The simulator requires the agent to HOLD the object to read it.
% The Model failed to generate:
% nonexecutable read(se) if object_is_grabbed(se)=false.

:- query
    ... read_executed=false ...
    maxstep: object_is_grabbed(novel_1000)=true & read_executed=true.

% Simplified Output Plan:
% 1. Walk(novel) -> 2. TurnTo(novel) -> 3. Open(novel)
% 4. Read(novel) -> 5. Grab(novel)
```

Analysis: The solver generated a plan where the agent reads the novel *before* grabbing it. This occurred because the model missed the specific constraint requiring grabbing. Consequently, the solver optimized the plan by deferring the `grab` action until after the `read` action, resulting in a "Missing Step" error during execution.

Missing Step due to overlooked environment states

```
:- objects
    cup_1000, cupboard_229 :: static_entity; ...

% Initial State (Defined correctly!)
% object_is_inside(cup_1000) = cupboard_229.
% object_is_open(cupboard_229) = false.

% Generated constraints for 'grab'
nonexecutable grab(se) if character_location \= object_location(se).
nonexecutable grab(se) if object_is_grabbable(se)=false.

% [MISSING CONSTRAINT]
% The model failed to generate the container accessibility rule:
% nonexecutable grab(se) if object_is_inside(se)=container
%                         & object_is_open(container)=false.

:- query
    ... object_is_inside(cup_1000)=cupboard_229 ...;
    maxstep: object_is_grabbed(cup_1000)=true.
```

```
% Simplified Output Plan:
% 1. Walk(cup)
% 2. Grab(cup)
% 3. Drink(cup)
```

Analysis: The model correctly recognized the cupboard as a relevant object but failed to enforce the containment physics. Without the explicit rule "you cannot grab an item inside a closed container," the solver treated the cup as directly accessible despite its location state being set to `cupboard_229`, causing a Missing Step error (failure to open the cupboard first).

B.2.6. SUMMARY

Our qualitative audit of the VirtualHome benchmark reveals distinct failure modes that differentiate LLM planners from the DecoVer framework. Direct LLM baselines primarily suffered from **Missing Step** errors, reflecting a fundamental disconnection from the physical constraints of the simulator. These failures manifested as *Spatial Neglect*, attempting to interact with distant objects without navigating and *Precondition Neglect*, ignoring affordances such as orientation or possession. Additionally, we identified a specific anomaly in high-capability models, where an artificially high "Hallucination Rate" was driven by the strict adherence to JSON formats. These models appended numerical suffixes to duplicate action keys (e.g., `WALK_1`), which the evaluation harness flagged as non-existent actions despite being semantically correct.

In contrast, DecoVer's failures were rarely due to probabilistic hallucination but rather stemmed from the rigid logic of the solver. The majority of failures resulted in empty plans because the solver determined the task was unsolvable under the provided constraints or the $\mathcal{BC}+$ program cannot be executed. These were caused by Over-Strict Constraints, syntax mismatches, or dependency failures where necessary utilities were absent from the signature. Conversely, when the generated constraints were too lax, the solver produced plans that were logically valid within $\mathcal{BC}+$ but physically invalid in the simulator, caused by the omission of specific containment or affordance rules.

C. Logically Extended Cases

To evaluate DecoVer's ability to generalize to novel mechanics beyond standard VirtualHome tasks, we designed four custom scenarios and implemented corrsponding executors and checkers based on the Embodied Agent Interface. To ensure a fair comparison, the baseline (plain LLM) prompts were augmented with a "safety heads-up" or brief functional description where appropriate.

We provided detailed explanations of the extended cases along with error analysis on these cases.

C.1. Cases

The four test categories are defined as follows:

Fix an Appliance. We introduce a new action, `FIX(obj)`, which transitions an object with the `Appliance` property from `BROKEN` to `FUNCTIONAL`.The critical logical constraint is safety: `FIX` requires the appliance to be `PLUGGED_OUT` before execution. Failure to unplug the device constitutes a safety violation.

Observe Small Text. We introduce the action `OBSERVE(obj)`. While this action has no physical side effects on the object, it requires a global environmental precondition: The room must be lit up to observe small text. This tests the agent's ability to link local object interactions with global state variables.

Hide Treasure. We introduce the actions `LOCK(obj1,obj2)` and `UNLOCK(obj1,obj2)` for containers with the `HAS_LOCK` property. These actions govern the transition between `LOCKED` and `UNLOCKED` states. The logic enforces mutual exclusion constraints: an object cannot be opened or closed while `LOCKED`, and conversely, it cannot be locked or unlocked while `OPEN`.

Make a Sandwich. The task requires the agent to stack multiple ingredients in a specific vertical order. Success depends on the model's ability to verify that the support surface (the previous ingredient) is available and correctly positioned before placing the next item.

C.1.1. TASK: FIX AN APPLIANCE

We define the escalating complexity of the "Fix" tasks through the following hierarchy. In all cases, the core logical constraint is that an appliance must be `PLUGGED_OUT` before the `FIX` action can be applied.

Level 1: Case 1024_19
Scenario: A broken TV is present. It is already `PLUGGED_OUT` and `SWITCHED_OFF`.
Task: Fix the TV.
Description: Base case. The environment is already in a safe state, so the model simply needs to identify the `FIX` action.

Level 2: Case 1024_33
Scenario: Same as Level 1 (TV is broken and unplugged).
Task: Fix the TV and turn it on.
Description: The model must sequence the repair followed by the restoration of power. The catch is to turn on the TV after it is fixed instead of the contrary.

Level 3: Case 1024_20
Scenario: A broken TV is currently `PLUGGED_IN` and `SWITCHED_ON`.
Task: Fix the TV.
Description: The model faces an immediate safety hazard. And must insert preparatory actions (`SWITCHOFF` → `PLUGOUT`) before executing `FIX`.

Level 4: Case 1024_34
Scenario: A broken TV is currently `PLUGGED_IN` and `SWITCHED_ON`.
Task: Fix the TV and ensure it is working (On/Plugged In) at the end.
Description: This combines the constraints of Levels 2 and 3. The model must first removing power to satisfy safety conditions, then restore power to satisfy goal conditions (On → Off → Unplug → Fix → Plug In → On).

Level 5: Cases 1024_21, 1024_22
Scenario: Two distinct broken appliances (e.g., a TV and a Lamp) are both `PLUGGED_IN`.
Task: Fix both appliances.
Description: The model must manage the state of multiple objects simultaneously.

C.1.2. TASK: OBSERVE SMALL TEXT

In the "Observe" scenarios, the complexity stems from the dependency on global environmental states (lighting) rather than just the object's intrinsic state.

Level 1:Cases 1024_23, 1024_24
Scenario: A note (or map) is present in the dark room.
Task: Observe the note/map.
Description: This establishes the baseline behavior. The model must derive two distinct preconditions: a local spatial constraint of facing the target and a global environmental constraint of lighting up the room.

Level 2: Case 1024_25
Scenario: A note is present, but the room lamp is currently `OFF` and `PLUGGED_OUT`.
Task: Observe the note.
Description: This level complicates the environmental precondition.

Level 3: Cases 1024_26, 1024_35
Scenario: The note is hidden inside a `CLOSED` cabinet. The lamp is `OFF` and `PLUGGED_OUT`.
Task: Observe the note.
Description: The most complex variant, combining visibility with physical accessibility.

C.1.3. TASK: HIDE TREASURE

The "Treasure" scenarios introduce the LOCK/UNLOCK primitives. Beyond simple state changes, the agent is faced with a hidden restriction of only being able to hold one item in one hand.

Level 1: Case 1024_27

Scenario: Two pieces of treasure and one key are present and accessible. The chest is currently unlocked.
Task: Put both treasures inside the chest and lock it.
Description: The primary challenge here is inventory management. The character has only two hands, but the task involves interacting with three distinct items (two treasures + one key). The model must plan a valid sequence that involves picking up treasures, placing them, and only *then* retrieving the key to perform the final lock, without exceeding its carrying capacity.

Level 2: Case 1024_28

Scenario: Same as Level 1, but the chest is initially LOCKED and CLOSED.
Task: Put both treasures inside the chest and lock it.
Description: The model cannot simply insert the items; it must first perform (UNLOCK $\rightarrow$ OPEN), insert the items, and then execute (CLOSE $\rightarrow$ LOCK). It tests the mutual exclusion logic: knowing that one cannot open a locked chest, nor lock an open one.

Level 3: Case 1024_29

Scenario: The two treasures are locked inside the chest.
Task: Grab both treasures in both hands
Description: The reverse of the standard task. This tests goal regression: to GRAB an item, it must be accessible; to be accessible, the container must be OPEN; to be open, it must first be UNLOCKED.

Level 4: Cases 1024_31, 1024_32

Scenario: The treasures (Case 31) or the key (Case 32) are not visible initially but hidden inside *other* closed containers in the room.
Task: Hide treasures in the chest and lock it.
Description: These cases maximize planning depth. The model cannot interact with the main chest immediately. It must first identify and navigate to auxiliary containers to collect the necessary components before bringing them to the central chest.

A twist: Case 1024_30

Scenario: The chest is already LOCKED and CLOSED.
Task: Execute the actions CLOSE and LOCK.
Description: This presents a unique "Redundant Execution" trap. Since the goal explicitly demands the *execution* of the actions (Action Goal), the model must first UNLOCK and OPEN the chest solely to create the necessary preconditions to validly execute CLOSE and LOCK again.

C.1.4. TASK: MAKE A SANDWICH

This task focuses on **Geometric Composition**. The model must strictly adhere to a vertical stacking order (e.g., Bread $\rightarrow$ Bacon $\rightarrow$ Cheese $\rightarrow$ Bread). The complexity scales based on the distribution of ingredients and the agent's carrying capacity relative to the required components.

Level 1: Case 1024_3

Scenario: Two pieces of bread and a plate are present and accessible.
Task: Put bread on the plate, then put the second bread on the first bread.
Description: The base case for geometric reasoning. The agent must respect the physical support hierarchy: the plate supports Bread 1, and Bread 1 supports Bread 2.

Level 2: Case 1024_4

Scenario: Three ingredients (Bread, Bacon, Bread) are stored inside a single closed cabinet.
Task: Make the sandwich.

Description: This adds a retrieval step. The model must identify the container, open it, and transfer items to the workspace.

Level 3: Case 1024_5

Scenario: Three ingredients are scattered across two separate closed cabinets.
Task: Make the sandwich.
Description: The model must visit distinct locations to gather components, testing whether it efficiently clears one source before moving to the next.

Level 4:Cases 1024_6, 1024_36, 1024_37

Scenario: Four ingredients are required. Three are in one container, one is in another.
Task: Make the sandwich.
Description: This is the test for inventory management. Since 4 items are needed but only 2 can be held at once, it is physically impossible to gather all ingredients from one location in a single pass.

C.2. Error Analysis

We provide detailed error cases with LLM-Planners and DecoVer.

C.2.1. LLM-PLANNER ERRORS

Fix An Appliance

```
"actions": [
    "[WALK] <television> (101)",
    "[FIX] <television> (101)",       <-- ERROR: Invalid State
    "[SWITCHOFF] <television> (101)",
    "[PLUGOUT] <television> (101)"
]
```

Analysis: The simulator requires an appliance to be and plugged out *before* it can be repaired. The model attempts to execute `FIX` immediately upon arrival, violating the safety condition despite an additional heads-up in the input prompt.

Observe Small Text

```
"actions": [
    "[WALK] <lamp> (102)",
    "[PLUGIN] <lamp> (102)",
    "[SWITCHON] <lamp> (102)",
    "[WALK] <closet> (100)",
    "[OPEN] <closet> (100)",
    "[WALK] <note> (101)",
    "[OBSERVE] <note> (101)" <-- ERROR: Missing TurnTo
]
```

Analysis: In this task, the problem definition included an explicit goal requiring the agent to be in a `FACING` relation with the note. Despite this explicit state requirement serving as a clue, the model neglected the necessary `TURNTO` action. The planner correctly navigated the complex environmental setup (lighting the room, opening the storage) but failed the final interaction by conflating the semantic intent of `OBSERVE` with the physical requirement of orientation.

Hide Treasure

```
"actions": [
    "[WALK] <table>", "[GRAB] <key>", "[WALK] <treasure_chest>",
    % Agent is holding the key, but tries to put gold (which is not held) into the chest
    "[PUTIN] <treasure_gold> <treasure_chest>" <-- ERROR: Not Holding
]

"actions": [
    ... "[OPEN] <treasure_chest>",
    % Agent assumes gold is reachable from the chest's location
    "[GRAB] <treasure_gold>" <-- ERROR: Not Close (Missing Walk)
]

% Initial State: Chest is CLOSED and LOCKED
"actions": [
    "[WALK] <treasure_chest>",
    % Goal requires Closed/Locked state; model blindly executes action
    % without checking if the state is already satisfied.
    "[CLOSE] <treasure_chest>" <-- ERROR: Cannot close locked object
]
```

Analysis: The "Hide Treasure" task rigorously stresses the planner's ability to manage prerequisites. In the first case, the model forgets to acquire the content (the gold) first. In the second case, it fails the spatial constraint: it treats the gold as locally accessible immediately after opening the chest, ignoring that the gold is located elsewhere. In the third case, the task requires executing `CLOSE` and `LOCK` but the chest is closed and locked to begin with. Driven by the textual goal, the model attempts to close an already locked chest.

Make a Sandwich

```
"actions": [
    "[GRAB] <toast_top> (201)",  % Hand 1 Occupied
    ...
    "[GRAB] <cheese> (203)",     % Hand 2 Occupied
    % Agent attempts to grab a third item without freeing a hand
    "[GRAB] <lettuce> (202)"     <-- ERROR: No Free Hand
]

"actions": [
    "[WALK] <Plate> (3003)",
    "[PUTBACK] <FoodToastBottom> (20) <Plate> (3003)",
    % Model assumes Bacon is accessible from the Plate's location
    "[GRAB] <Bacon> (23)"        <-- ERROR: Not Close (Missing Walk)
]
```

Analysis: In the first case, the model fails the inventory constraint by grabbing a third ingredient before putting the first two down. The second case demonstrates a consistant error of assuming objects are close and negelecting a second walk action.

C.2.2. DECOVER ERRORS

Since Logically Extended Cases primarily introduce hard-to-identify preconditions, DecoVer's errors are highly associated to missing restrictions.

Fix An Appliance(DecoVer)

```
% Generated constraints
nonexecutable fix(se) if (object_is_broken(se)=false).
nonexecutable fix(se) if character_location \= object_location(se).
% [MISSING CONSTRAINT]
% The model failed to identify the restriction associated to safety:
% nonexecutable fix(se) if object_is_plugged_in(se)=true.
```

Simplified Solver Output:

```
1. [WALK] <light>
2. [PLUGOUT] <light>
3. [FIX] <light>       (Success: Serendipitous ordering)
4. [WALK] <television>
5. [SWITCHOFF] <television>
6. [FIX] <television>  (FAIL: TV is still plugged in)
7. [PLUGOUT] <television>
```

Analysis: Because the LLM failed to discover the constraint "Appliance must be unplugged before fixing," the solver viewed `PLUGOUT` and `FIX` as independent actions. For the first appliance (Light), the solver *accidentally* generated the correct order (Plugout → Fix). For the second (TV), it generated an equally valid logical sequence (SwitchOff → Fix) which violated the hidden simulator physics. This proves that without explicit restrictive constraints, DecoVer's success can sometimes be attributed to solver randomness rather than robust planning.

Hide Treasure (DecoVer)

Task: Hide Treasure

```
% Generated constraints for 'putin'
nonexecutable putin(se,se1) if character_location \= object_location(se1).
nonexecutable putin(se,se1) if object_is_openable(se1)=false.
% ... [other constraints] ...

% [MISSING CONSTRAINT]
% The model failed to generate the critical physics rule:
% nonexecutable putin(se,se1) if object_is_open(se1)=false.

:- query ... inside(gold, chest) & inside(silver, chest) & locked(chest) ...
```

Execution Trace (Solver Output):

```
1. [WALK] <gold>
2. [GRAB] <gold>
3. [WALK] <chest>
4. [CLOSE] <chest>       (Solver satisfies 'closed' goal early)
5. [PUTIN] <gold> <chest> (FAIL: Cannot put item into closed chest)
```

Analysis: This failure is driven by the solver's optimization logic colliding with missing physics. The task requires the chest to be closed *eventually*. Since the model failed to state that "Insertion requires an Open container," the solver may execute `CLOSE` before `PUTIN`, resulting in an execution failure when the simulator enforced the physical reality.

D. Temporally Extended Cases

While the previous scenarios evaluated the model's ability to infer novel causal mechanisms (Depth), this section evaluates the model's ability to maintain logical consistency over long execution horizons (Length). These tasks are characterized by their repetitive nature and high step counts, designed to test models' state tracking capabilities.

D.1. Cases

We categorize these long-horizon tasks into five distinct domains:

1. **Store Drinks.** This category involve sorting a large volume of objects into specific containers based on their properties.
2. **Do the Dishes.** This category focuses on state transition cycles. The agent must process utensils through a strict pipeline: Clean up dirty kitchenware and store them away. Some variants extend this pipeline to include waste disposal.
3. **Set Up Family Theatre.** This category require a diverse set of interactions to configure a room for a specific activity. The agent must orchestrate object manipulation (placing snacks/discs), device control (operating the DVD player/TV), and environmental control (lighting), testing the coordination of distinct action types.
4. **Prepare For Work.** This category simulate daily routines involving the gathering of disparate objects (laptop, papers, coffee) to a central workspace. Uniquely, these cases often enforce a specific **Terminal Posture Constraint**, for instance, requiring the charcter to be `SITTING` in the end.
5. **Set Up Dinner Party.** The most heterogeneous category, combining elements of the previous tasks.

D.1.1. Task: Store Drinks

These scenarios focus on high-volume sorting and repetitive state manipulation. The difficulty lies in maintaining the correct mapping between object types and destination containers over many steps.

Level 1: Case 1025_1
Scenario: Nine drinks total (three types, three of each) are scattered in the environment. Three distinct destination containers are designated.
Task: Sort each drink type into its specific container.
Description: This tests the model's ability to maintain a classification rule over a long execution horizon.

Level 2: Case 1025_2
Scenario: Nine drinks are present.
Task: Place eight drinks on the table, but keep the ninth one in the character's hand.
Description: The model must process the bulk of the items but inhibit the "Place" action for the final item.

Level 3: Case 1025_3
Scenario: Five drinks are present.
Task: Put three of the drinks in the freezer and two in the garbage bin.
Description: The model must manage two distinct destination logic streams other than one.

Level 4: Case 1025_4
Scenario: Same as Level 1, but all drinks are initially stored inside a single box.
Task: Sort the drinks into their respective containers.
Description: This adds an unpacking layer. The model cannot simply navigate to the drinks; it must first locate the source container, open it, and then proceed with the sorting.

D.1.2. TASK: DO THE DISHES

These scenarios focus on sequential cleaning pipelines. The agent must successfully transition objects through specific state changes (e.g., `DIRTY` → `CLEAN`) before storing them, strictly adhering to the "Wash before Store" protocol.

Level 1: Case 1025_5
Scenario: Five utensils are initially `DIRTY`.
Task: Wash all utensils (making them `CLEAN`) and store them in the cabinet.
Description: The basic cleaning pipeline. The agent must identify the dirty state and apply `WASH` before the final placement.

Level 2: Case 1025_7
Scenario: A mixed collection of objects: three dirty utensils and two pieces of garbage.
Task: Clean and store the utensils; dispose of the garbage in the bin.
Description: The agent must distinguish between "cleanable" objects (utensils) and "disposable" objects (garbage). It tests the ability to branch the pipeline as in the Basic version.

Level 3: Case 1025_6
Scenario: Same as Level 1 (5 dirty utensils), but the storage destination is a specialized Disinfection Cabinet.
Task: Wash utensils, store them, and finally switch on the disinfection cabinet.
Description: This extends the pipeline. After the repetitive cycle is complete, the model must perform a final state change on the container itself with `SWITCHON`.

Level 4: Case 1025_8
Scenario: Multiple dirty utensils are present.
Task: Load all utensils into the dishwasher, close it, and switch it on.
Description: Simpler version of Level 1 as the `WASH` action may be ommitted.

D.1.3. TASK: SET UP HOME THEATRE

These scenarios represent "End-State Configuration" tasks. Here the agent must achieve a specific combination of diverse states (On, Off, Inside, OnTop, Facing) for different entities.

Level 1: Case 1025_9, 1025_10, 1025_11
Scenario: A living room environment with scattered food, a TV, a DVD player, a disc, and lights.

Task: Configure the room for a movie night. Specifically: A television is ON and PLUGGED_IN; A Disc is INSIDE the DVD player and the DVD player is CLOSED; All snacks are ON the table; The light is OFF, and the character is FACING the TV.
Description: This is a checklist-style task. The model must sequence actions to satisfy distinct states and relations with different Actions.

Level 2: Case 1025_12
Scenario: Same as Level 1.
Task: Same as Level 1, but with the additional constraint that the character must be holding the REMOTE_CONTROL at the end.
Description: This introduces a specific inventory reservation.

D.1.4. TASK: PREPARE FOR WORK

These scenarios simulate a daily routine. The core challenge is sequencing: the agent must gather materials and configure electronics *before* adopting the final posture, as sitting typically marks the end of the mobile phase.

Level 1: Basic (Case 1025_13)
Scenario: Laptop and coffee mug are initially away from the desk. A monitor is present on the desk.
Task: Bring the laptop and mug to the desk, switch on both the laptop and monitor, and finally WATCH an object to simulate work.
Description: Basic case of setting up workspace.

Level 2: Final posture constraint(Case 1025_14, 1025_15, 1025_16)
Scenario: Same as Level 1.
Task: Same as Level 1, but the character must be SITTING on the chair at the end.
Description: This introduces an execution order constraint. The agent must perform all navigation and manipulation tasks *before* sitting down.

D.1.5. TASK: HOLD DINNER PARTY

This final category serves as a comprehensive integration test, combining the sanitation logic of "Do the Dishes," the environmental staging of "Home Theatre," and the terminal posture constraints of "Prepare for Work." Case 1025_17 requires the model to retrieve five dirty utensils from a cabinet, wash them one by one, and set them on the dining table. Case 1025_18 is a complex staging task that requires decorating a Christmas tree with lights and switching them on, serving food, and finally having the character sit at the table. This merges electrical activation goals with geometric placement and posture constraints. Cases 1025_19 and 1025_20 introduce stricter terminal constraints. Case 19 requires the character to end the task seated while holding specific party items, and Case 20 scales the complexity by increasing the number of christmas lights to put up.

D.2. Errors

D.2.1. LLM-PLANNER ERRORS

The introduction of these cases does not yield fundamentally *new* error categories; the failure modes remain consistent with previous sections. However, the *frequency* and *severity* of these errors increase with LLM-Planners.

Do The Dishes

```
"actions": [
    "[WALK] <bowl> (101)",
    "[GRAB] <bowl> (101)",
    "[WALK] <disinfection_cabinet> (200)",
    "[WASH] <bowl> (101)",

    % Action 1: Put the bowl in the cabinet (Implicitly releases hold)
    "[PUTIN] <bowl> (101) <disinfection_cabinet> (200)",

    % Action 2: Attempt to release the bowl again
    "[RELEASE] <bowl> (101)" <-- ERROR: Not Holding
]
```

Analysis: The `PUTIN` action implicitly transfers possession of the object from the agent to the container, clearing the agent's inventory. But the mode redundantly issues a `RELEASE` command. This is a classic example of *State Tracking Failure* where the model fails to update its internal belief state after a composite interaction.

Set Up Family Theatre

```
"actions": [
    ...
    "[WALK] <dvd_player> (101)",

    % CRITICAL ERROR (Step 4): Agent prematurely closes the player
    "[CLOSE] <dvd_player> (101)",

    ... (15 intermediate steps involving food/lights) ...

    "[GRAB] <disc> (102)",
    "[WALK] <dvd_player> (101)",

    % FAILURE (Step 20): Agent attempts insertion into a CLOSED object.
    % Simulator resolves this as placing the disc ON TOP of the player.
    "[PUTBACK] <disc> (102) <dvd_player> (101)"
]
```

Analysis: This failure illustrates the difficulty of maintaining state consistency over long horizons (20+ steps). Early in the trajectory, the agent executed `CLOSE` which unintentionally invalidated the physical precondition for the final goal of inserting the disc. When the agent returned to the DVD player 15 steps later, it failed to verify the state or re-open the device. Consequently, the `PUTBACK` command collided with the closed geometry of the player, resulting in the disc resting **ON** the surface rather than **INSIDE** the slot.

Prepare For Work

```
"actions": [
    "[WALK] <chair> (105)",

    % CRITICAL ERROR: Agent sits immediately to satisfy the goal
    "[SIT] <chair> (105)",

    % The agent is now sitting, but a series of other goals aren't satisfied yet.
    "[WALK] <computer> (100)" <-- ERROR: Cannot walk while sitting
    ...Subsequenting actions ommitted due to early failure.
]
```

Analysis: The task requires the agent to perform various setup actions (plugging in devices, grabbing the mouse) and *end* the episode in a seated position. The LLM, failing to reason about the temporal constraints of mobility, prioritized the `SIT` action immediately upon reaching the chair. This resulted in a physical deadlock: the `SIT` action changed the agent's posture state to `SITTING`, rendering the subsequent `WALK` actions impossible. The model failed to recognize that `SIT` is a terminal action regarding navigation and must be scheduled strictly after all other tasks are complete.

D.2.2. DECOVER ERRORS

In contrast, DecoVer shows relative stability in these long-horizon tasks, as the temporal complexity is offloaded entirely to the $\mathcal{BC}+$ solver. However, while the *temporal* depth is handled by the solver, the *breadth* of the action space imposes a

significant cognitive load on the ground rules and restrictions generation phases. weaker models may still induce failures through under-specification when the workload for generating ground rules and restrictions becomes excessive.

Set Up Home Theatre(DecoVer)

Generated Constraint Block

```
% [CRITICAL ERROR] Copied directly from prompt example
nonexecutable grab(se) if (object_is_onto(se1)=se & is_too_high(se1)=true).

% Standard constraints follow...
nonexecutable close(se) if character_location \= object_location(se).
...
```

Analysis: This error represents a failure of *context isolation*. The constraint regarding `is_too_high` was present in the system instructions as a few-shot example to demonstrate complex logic. The model, rather than generalizing the *pattern*, indiscriminately copied the *content* of the example into the active constraint set. Since the predicate `is_too_high` is not defined in the current problem's signature, this typically results in a syntax error.

Hold Dinner Party (DecoVer)

```
% [CRITICAL ERROR] Hallucinated Requirements:
nonexecutable switchon(se) if (character_is_sitting=false).
nonexecutable switchoff(se) if (character_is_sitting=false).

% Standard Physics: Cannot walk while sitting
nonexecutable walk(se) if character_is_sitting=true.
```

Analysis: This failure stems from a semantic hallucination where the LLM overly restricts the affordances of the agent. This creates a logical deadlock for the solver. To switch on a wall light, the agent must be at the light's location. However, if there is no chair explicitly at that location, the agent cannot satisfy the `character_is_sitting` precondition, rendering the `SWITCHON` action globally executable but locally impossible.

E. Error Analysis On MCP and Hanoi

In this section, we provide an analysis on the error cases of the MCP and Hanoi domains.

E.1. Variants

Details of the MCP variants are shown below.

I. Standard Settings

- **Std1:** The traditional formulation with 3 missionaries and 3 cannibals. The boat holds 2 people, and missionaries must never be outnumbered by cannibals on either bank.
- **Std2:** Missionaries and cannibals may exchange hats.

II. Entity-Based variants These variants include special entities that induce more strict constraints or ease existing constraints.

- **Ent1 (Two Rowers):** Only one missionary and one cannibal can row.
- **Ent2 (Oars):** The boat requires a resource (an oar) to cross. Oars are distributed across banks, and the number of people crossing cannot exceed the number of available oars at the departure location.
- **Ent3 (Big Cannibal):** Introduces a "Big Cannibal" that must cross alone.
- **Ent4 (Big Cannibal Small Missionary):** Introduces a constraint where a specific "Big Cannibal" cannot be left alone with a specific "Small Missionary," adding a pairwise exclusion rule to the standard safety constraints. Note that the capacity constraint in **Ent3** does not apply here.

- **Ent5 (Jesus):** Introduces a special missionary who possesses the unique ability to `Walk` on water.

III. Action-Based variants These variants alter the transitions avaliable.

- **Act1 (Bridge):** Adds a bridge that allows people to `Walk` across the river directly.
- **Act2 (Conversion)**: 3 missionaries can transform a cannibal into a missionary if they are left alone on the same side of the bank.
- **Act3 (Island):** There is an island in the middle of the river where the boat can go to.

E.2. LLM-Planner Errors in MCP

In MCP, the basic rule where Cannibals may not outnumber Missionaries at any location is frequently violated even with high-capability models.

Safety Constraint Violation (Oars)

```
 I need to solve this MCP problem with an oar constraint. Let me analyze:

- 3 missionaries, 3 cannibals need to cross from left to right
- Boat capacity: 2 people max
- Oar constraint: 1 oar on each bank initially. Need 1 oar for 1 person, 2 oars for 2 people
- Safety: cannibals <= missionaries (when missionaries > 0) on both banks and boat

Let me track: [missionaries, cannibals, oars] for each location.

Initial state: Left_bank: [3,3,1], Right_bank: [0,0,1]

First, the boat is at left bank with 1 oar. Only 1 person can cross initially.

1. Start: Left[3,3,1], Right[0,0,1] - boat at left
2. 1 cannibal crosses right (picks up right oar, now boat has 2 oars): Left[3,2,0], Right[0,1,0]
3. 1 cannibal returns left: Left[3,3,0], Right[0,0,0] - boat now has 2 oars
4. 2 cannibals cross right: Left[3,1,0], Right[0,2,0]
5. 1 cannibal returns left: Left[3,2,0], Right[0,1,0]
6. 2 missionaries cross right: Left[1,2,0], Right[2,1,0]
7. 1 missionary, 1 cannibal return left: Left[2,3,0], Right[1,0,0]
8. 2 missionaries cross right: Left[0,3,0], Right[3,0,0]
9. 1 cannibal returns left: Left[0,4,0]... wait, that's wrong.

Let me redo this more carefully:

1. Left[3,3], Right[0,0] - Start
2. 1C crosses → Left[3,2], Right[0,1] (1 oar used, picks up 2nd oar)
3. 1C returns → Left[3,3], Right[0,0] (now have 2 oars)
4. 2C cross → Left[3,1], Right[0,2]
5. 1C returns → Left[3,2], Right[0,1]
6. 2M cross → Left[1,2], Right[2,1]
7. 1M,1C return → Left[2,3], Right[1,0] - Invalid! (2<3 on left)

Let me try again:
6. 2M cross → Left[1,2], Right[2,1]
7. 1C returns → Left[1,3], Right[2,0] - Invalid!

Different approach:
6. 2M cross → Left[1,2], Right[2,1]
This is the failing step. The model fails to correct this despite self-refinement.

7. 1M returns → Left[2,2], Right[1,1]
8. 2M cross → Left[0,2], Right[3,1]
9. 1C returns → Left[0,3], Right[3,0]
10. 2C cross → Left[0,1], Right[3,2]
11. 1C returns → Left[0,2], Right[3,1]
12. 2C cross → Left[0,0], Right[3,3]
```

Analysis: The model fails to look ahead to the consequences of moving the majority of missionaries. By moving 2 missionaries to the right bank in Step 6, it leaves a single missionary alone with two cannibals on the left bank.

Losing track of entity location(Two Rowers)

```
I need to solve the Missionaries and Cannibals problem where only one missionary and one cannibal can row the boat.

Let me denote states as [missionaries, cannibals] on each bank, and track the boat position.

Initial: Left [3,3], Right [0,0], Boat on Left
Goal: Left [0,0], Right [3,3], Boat on Right

Constraints:
- Boat capacity: max 2 people
- Cannibals <= Missionaries on each bank (if missionaries > 0)
- Cannibals <= Missionaries on boat (if missionaries > 0)
- Only 1 missionary (M1) and 1 cannibal (C1) can row
- Someone who can row must be on the boat

Let me work through this:

1. Start: L[3,3], R[0,0] - boat left
2. Send 1M(rower) + 1C → L[2,2], R[1,1] - boat right
3. Return 1M(rower) → L[3,2], R[0,1] - boat left
4. Send 2C (one is rower) → L[3,0], R[0,3] - boat right
5. Return 1C(rower) → L[3,1], R[0,2] - boat left
6. Send 2M (one is rower) → L[1,1], R[2,2] - boat right
7. Return 1M+1C (rowers) → L[2,2], R[1,1] - boat left
8. Send 2M (one is rower) → L[0,2], R[3,1] - boat right
9. Return 1C(rower) → L[0,3], R[3,0] - boat left
10. Send 2C (one is rower) → L[0,1], R[3,2] - boat right
11. Return 1C(rower) → L[0,2], R[3,1] - boat left
12. Send 2C (one is rower) → L[0,0], R[3,3] - boat right
```

Analysis: The model lost track of the location of the rowers in step 7. There was at most one rower on the right bank by step 6, so the action in step 7 was invalid.

Abrupt Failure (Big Cannibal)

```
 ```json
[
    {"Left_bank":[3,2,1],"Right_bank":[0,0,0]},
    {"Left_bank":[3,3,0],"Right_bank":[0,0,1]},
    {"Left_bank":[2,2,0],"Right_bank":[1,1,1]},
    {"Left_bank":[3,2,0],"Right_bank":[0,1,1]},
    {"Left_bank":[3,3,0],"Right_bank":[0,0,1]},
    {"Left_bank":[1,2,0],"Right_bank":[2,1,1]},
    {"Left_bank":[2,2,0],"Right_bank":[1,1,1]},
    {"Left_bank":[0,2,0],"Right_bank":[3,1,1]},
    {"Left_bank":[1,3,0],"Right_bank":[2,0,1]},
    {"Left_bank":[1,2,0],"Right_bank":[2,1,1]},
    {"Left_bank":[1,3,0],"Right_bank":[2,0,1]},
    {"Left_bank":[0,3,0],"Right_bank":[3,0,1]}
]
 ```
```

Analysis: The task isn't completed by the final step, and an additional cannibal emerged in step 2. The big cannibal is one of the three cannibals but four cannibals add up after the execution of step two.

E.3. LLM-Planner Errors in Hanoi

We omit detailed qualitative case studies for the Towers of Hanoi domain for two reasons. First, the failure modes for this task are highly homogeneous compared to the errors observed in VirtualHome or MCP. Baseline failures almost exclusively manifest as state tracking failures: the model either attempts to move a disk that is currently occluded or violates the fundamental size constraint by placing a larger disk atop a smaller one.

Second, the textual representation of these errors is opaque to the reader. Unlike the semantically rich actions of embodied planning, a Hanoi plan consists of a long sequence of abstract operations. Isolating a specific error within a 30-step trace requires the reader to mentally simulate the entire game state up to that point—a process that is cognitively demanding and offers little insight beyond the aggregate failure rates reported in the main text. For anyone who concerns, we will provide a checker for Hanoi sequences in the code repository, although this is easy to implement.

E.4. DecoVer Errors in MCP

DecoVer Errors in MCP primarily stems from over-strict restrictions or hallucinations in ground rules.

Over Strict Restrictions

```
%added constraints:
nonexecutable cross(from,to) if (N1 + N2 < 1)
& N1 = numNormalCannCrossing(from,to) & N2 = numNormalMissCrossing(from,to).
nonexecutable cross(from,to) if (N1 < 0) & N1 = numNormalCannCrossing(from,to).
nonexecutable cross(from,to) if (N2 < 0) & N2 = numNormalMissCrossing(from,to).
nonexecutable cross(from,to) if (N1 + N2 > 2)
& N1 = numNormalCannCrossing(from,to) & N2 = numNormalMissCrossing(from,to).
nonexecutable cross(from,to) if (N2 > 0 & N1 > N2)
& N1 = numNormalCannCrossing(from,to) & N2 = numNormalMissCrossing(from,to).
nonexecutable cross(from,to) if boat_is_at \= from.
impossible numNormalMissAt(loc) < 0.
impossible numNormalCannAt(loc) < 0.
[Wrong Restriction]
impossible (N1 + N2 > N3) & N1=numNormalCannAt(loc) & N2=1 & N3=numNormalMissAt(loc).
noconcurrency.
```

Analysis: The rule of "Missionaries may not be outnumbered by Cannibals" implies a non-zero count of Missionaries. The modeled rule prohibits a state with zero missionaries and non-zero cannibals, leading to a No Solution Error as any valid transition sequence requires zero missionaries at one bank at some time.

Hallucinations in Ground Rules

```
%Ground Rules:
cross(from,to) decrements numNormalCannAt(from) by numNormalCannCrossing(from,to).
cross(from,to) increments numNormalCannAt(to) by numNormalCannCrossing(from,to).
cross(from,to) decrements numNormalMissAt(from) by numNormalMissCrossing(from,to).
cross(from,to) increments numNormalMissAt(to) by numNormalMissCrossing(from,to).
%The following rules are harmful.
cross(from,to) decrements numNormalCannAt(from) by 0.
cross(from,to) decrements numNormalMissAt(from) by 0.
cross(from,to) increments numNormalCannAt(to) by 0.
cross(from,to) increments numNormalMissAt(to) by 0.
cross(from,to) causes boat_is_at=to.
```

Analysis: The first four rules are correct. However, the model reasoned "Crossing some people from a to b doesn't decrease the head count at b". While this reasoning is correct but redundant, its realization in the form of ground rules render the problem unsolvable.

F. Ablation Studies

We provide specific error cases in our ablation experiments.

F.1. Removing Verification Modules

No Action Verification

```
// The selcted actions does not contain "OPEN",but the
target object is inside a container.
puton(static_entity,static_entity) :: exogenousAction;
switchon(static_entity) :: exogenousAction;
grab(static_entity) :: exogenousAction;
turnto(static_entity) :: exogenousAction;
walk(static_entity) :: exogenousAction;
run(static_entity) :: exogenousAction;
plugin(static_entity) :: exogenousAction;
...
0: object_is_inside(pot_1001)=freezer_289
...

0:  character_facing=void
character_location=start_point
grabbed_item_count=0 ...
```

```
ACTIONS:  walk(pot_1001)

1:  character_facing=void character_location=loc_pot_1001
   grabbed_item_count=0 ...

ACTIONS:  grab(pot_1001)

2:  character_facing=void character_location=loc_pot_1001 grab_executed grabbed_item_count=1 ...
```

Analysis: The action `OPEN` is omitted due to low direct relevance to the goals, as a result, the restriction of opening before grabbing is also absent, leading to a missing step error.

No Ground Rules Verification

```
%Ground Rules
open(se) causes object_is_open(se)=true.
grab(se) causes object_is_grabbed(se)=true.
grab(se) increments grabbed_item_count by 1.
putin(se,se1) causes object_is_inside(se)=se1.
putin(se,se1) causes object_is_grabbed(se)=false.
putin(se,se1) decrements grabbed_item_count by 1.
turnto(se) causes character_facing=se.
switchon(se) causes object_is_switched_on(se)=true.
switchon(se) causes object_is_plugged_in(se)=true. #Wrong Rule
plugin(se) causes object_is_plugged_in(se)=true.
grab(se) causes grab_executed=true.
walk(se) causes character_location=object_location(se).
```

Analysis: The marked rule is illogical. Even though this rule doesn't harm the final execution, it should have been removed.

F.2. Tampering with Decomposition Mechanisms

All Actions and Objects in Flag Selection

```
character_facing :: inertialFluent(static_entity);
character_is_inside :: inertialFluent(static_entity);
character_location :: inertialFluent(location);
grabbed_item_count :: additiveFluent;
lookat_executed :: inertialFluent(boolean);
object_is_grabbable(static_entity) :: inertialFluent(boolean);
object_is_grabbed(static_entity) :: inertialFluent(boolean);
object_is_inside(static_entity) :: inertialFluent(static_entity);
object_is_lookable(static_entity) :: inertialFluent(boolean);
object_is_readable(static_entity) :: inertialFluent(boolean);
object_location(static_entity) :: inertialFluent(location);
read_executed :: inertialFluent(boolean).
(The flag "object_is_openable" was not selected, but was required in the initial state .)
......
0: object_is_openable(nightstand_100)=true & ......
```

Analysis: Two flags are necessary to model an openable container: "object_is_openable" models the openable property and "object_is_open" models the corresponding state. In this case, the first flag is missing, leading to a $\mathcal{BC}+$ solver failure.

w/o Flag Grouping in Ground Rule Generation

```
%Ground Rules
open(se) causes object_is_open(se)=true.
putin(se,se1) causes object_is_inside(se)=se1.
grab(se) causes object_is_grabbed(se)=true.
switchon(se) causes object_is_switched_on(se)=true.
plugin(se) causes object_is_plugged_in(se)=true.
walk(se) causes character_location=se.
turnto(se) causes character_facing=se.
walk(se) causes character_location=object_location(se).

//No rule modifies "grabbed_item_count", the count is always zero.
....
nonexecutable putin(se,se1) if character_location \= object_location(se1).
nonexecutable putin(se,se1)  if object_is_grabbed(se)\=true.
nonexecutable putin(se,se1) if grabbed_item_count<=0.
```

```
nonexecutable putin(se,se1) if (se=void | se1=void).
```

Analysis: The ground rules modifying "grabbed_item_count" are absent. This poses a problem as grabbed_item_count is zero by default, that means the action PUTIN will always be prohibited, resulting in no solution.

w/o Decomposition in Restriction Generation

```
nonexecutable grab(se) if character_location \= object_location(se).
nonexecutable grab(se) if object_is_grabbable(se)=false.
nonexecutable grab(se)  if object_is_grabbed(se)\=false.
nonexecutable grab(se) if grabbed_item_count>=2.
...
0:object_is_inside(novel_1000)=nightstand_100...


    0:  character_facing=void character_location=start_point
    grabbed_item_count=0
    object_is_grabbable(novel_1000)
    object_is_grabbable(void) object_is_inside(nightstand_100)=void
    object_is_inside(novel_1000)=nightstand_100
    object_is_inside(void)=novel_1000
    object_is_lookable(void)
    object_is_openable(nightstand_100)
    object_is_openable(novel_1000)
    object_is_readable(novel_1000)
    object_location(nightstand_100)=loc_nightstand_100 object_location(novel_1000)=loc_novel_1000
    object_location(void)=start_point

 ACTIONS:  walk(novel_1000)

 1:  character_facing=void
    character_location=loc_novel_1000
    grabbed_item_count=0
    object_is_grabbable(novel_1000)
    object_is_grabbable(void)
    object_is_inside(nightstand_100)=void
    object_is_inside(novel_1000)=nightstand_100
    object_is_inside(void)=novel_1000
    object_is_lookable(void)
    object_is_openable(nightstand_100)
    object_is_openable(novel_1000)
    object_is_readable(novel_1000)
    object_location(nightstand_100)=loc_nightstand_100 object_location(novel_1000)=loc_novel_1000
    object_location(void)=start_point

 ACTIONS:  grab(novel_1000) //Missing Step Error.
```

Analysis: Without decomposing the restriction obtaining step into two phases, it is common for models to overlook complex restrictions. In this setting, all models missed out the restriction prohibiting grabbing an object in a container, leading to missing step errors.

G. Experiment Analysis on LLM+AL

In this section, we provide an in-depth analysis of the **LLM+AL** baseline performance on the standard VirtualHome dataset with 100 randomly selected cases. We first examine the *Solvability Rate* of the generated programs. We define a case as "Attempted" (or compilation successful) if the synthesized $\mathcal{BC}+$ program is syntactically valid and logically consistent enough for the solver to return *any* candidate action sequence, regardless of whether that sequence successfully completes the task in the end.

The two foundation models exhibited drastically different behaviors: textbfGemini-3-Pro-Preview demonstrated acceptable competence, which produced 98 solvable $\mathcal{BC}+$ programs. While **GPT-4o** struggled significantly with the strict syntactical and logical requirements of the representation, producing only 35 solvable $\mathcal{BC}+$ programs.

G.1. Error Analysis: Gemini-3-Pro-Preview

Although Gemini-3-Pro-Preview provided 98 valid outputs, it suffered from higher error rates than the model on its own. We categorize the errors into four distinct categories.

Incomplete Modelling. The most frequent failure mode involves the complete omission of navigation logic. In cases such as **137_1**, **180_1**, **504_2**, and **963_2**, the generated plan interacts with a target object without walking to the designated object first. Noteably, the inputs prompts to LLM+AL gave a direct heads-up to `WALK` to an object before interacting with it. Analysis of the $\mathcal{BC}+$ programs generated reveals that the action constant `WALK` and location states were not modeled in the initial signature, and never brought up in proceeding revision steps. The self-revision loop failed to correct this due to Circular Validation. The sample query generator assumes the provided modeling (however flawed) is ground truth and generates queries *within* that environment. If the initial modelling is flawed, the generator produces queries under that faulty assumption. Thus, the pipeline validates internal consistency but fails to challenge the external validity of the environmental itself.

Location Ambiguity. A subtle variation of spatial failure occurs when the model recognizes the need for modelling locations but misidentifies the granularity. In cases **323_2**, **509_2**, and **660_1**, the generated plan navigated the agent to a *Room* (e.g., `walk(bathroom)`) but failed to approach the specific *Object* (e.g., `sit(toilet)`). This stems from an initialization error where the model assigned the *Room's* location ID to the *Object* in the initial state definition, even after being given a heads-up that all objects have different locations. The solver, treating these distinct entities as spatially identical, successfully verified the precondition for interaction, leading to execution failures where the agent attempted to use an object from across the room.

Action Omission. Even when navigation is successful, the model consistently overlooks action preconditions. In cases **1083_1**, **15_2**, **221_2**, and **897_2**, the agent failed to execute `TurnTo` before `Watching` something. In these instances, the `TurnTo` action was either omitted entirely or present but without any groundrules or restrictions regarding it. Without explicit restrictions linking orientation to visibility (e.g., `nonexecutable watch if not facing`), the solver optimizes it out of the final plan,

Affordance Violation. Finally, output plans struggles to map actions to valid physical targets because the model fails to generate affordance properties (e.g., `LOOKABLE`, `TYPEABLE`). In case **454_2**, the plan applied `Watch` to a couch. The model failed to generate the `LOOKABLE` constraint, so the solver permitted the action; In case **87_2**, the plan attempted to `Type` on a desk for similar reasons. Also in case **662_1**, the agent attempted to `drink(water)` directly rather than from the cup, failing to link the drinking action to the container holding the liquid.

The following listings present the raw $\mathcal{BC}+$ programs generated by the LLM+AL baseline. We highlight the specific modeling deficits in the captions below each case.

Incomplete Modelling (Task: Answer Phone)

```
:- sorts
    entity; main_entity >> entity; static_entity >> entity; location.
:- objects
    character_65 :: main_entity;
    floor_202, ... phone_247, ... :: static_entity;
    loc_floor_202, ... loc_phone_247, ... :: location.
:- variables
    M :: main_entity; S :: static_entity; L :: location.
:- constants
    find(main_entity, static_entity) :: exogenousAction;
    walk(main_entity, location) :: exogenousAction;
    grab(main_entity, static_entity) :: exogenousAction;
    % ... [other actions omitted for brevity] ...
    holds_rh(main_entity, static_entity) :: inertialFluent(boolean);
    sitting(main_entity) :: inertialFluent(boolean);
    % [NOTE: 'loc' or 'location' fluent is MISSING entirely]

% Generated constraints
% 1. Grabbing causes holding
grab(M, S) causes holds_rh(M, S).
% ... [Constraint 10]
nonexecutable walk(M, L) if sitting(M).
% ... [Constraint 22]
impossible holds_rh(M, S1) & holds_rh(M, S2) & S1 \= S2.

noconcurrency.
:- query
    0: sitting(character_65) & ... & ~holds_rh(character_65, phone_247);
    maxstep: holds_rh(character_65, phone_247).
```

Analysis: The model fails to define a fluent for location (e.g., `loc(M)`). While `walk` is listed as a constant, there are

no causal laws linking it to a state change. Consequently, the solver generates a plan where the agent `grabs` the phone instantly without walking to the phone.

Location Ambiguity (Task: Sit on Toilet)

```
:- sorts
    entity; location; main_entity; static_entity;
    entity >> main_entity; entity >> static_entity.
:- objects
    character_65 :: main_entity;
    toilet_37, void :: static_entity;
    bathroom_1, home_office_319 :: location.
:- constants
    loc(main_entity) :: inertialFluent(location);
    character_on :: inertialFluent(static_entity);
    walk(location) :: exogenousAction;
    sit(static_entity) :: exogenousAction.

% Generated constraints
walk(L) causes loc(character_65)=L.
nonexecutable walk(L) if loc(character_65)=L.
sit(S) causes character_on=S.
nonexecutable sit(S) if character_on=S.

% [CRITICAL ERROR BELOW]
% 6. Sitting is not executable if not at the location of the static entity.
% Assuming toilet_37 is located in bathroom_1.
nonexecutable sit(toilet_37) if loc(character_65) \= bathroom_1.

noconcurrency.
:- query
    0: loc(character_65) = home_office_319 & character_on = void;
    maxstep: character_on = toilet_37.
```

Analysis: The model conflates the *Room* (`bathroom_1`) with the *Object Location*. Constraint 6 explicitly permits sitting on the toilet as long as the agent is inside the bathroom, ignoring the need to navigate specifically to the toilet itself.

Action Omission (Task: Read Novel)

```
:- sorts
    entity >> object; object >> main_entity; static_entity >> real_static_entity.
:- objects
    character_65 :: main_entity;
    table_355, novel_1000 ... :: real_static_entity;
    ...
:- constants
    walk(main_entity, location) :: exogenousAction;
    read_action(main_entity, real_static_entity) :: exogenousAction;
    lookat(main_entity, real_static_entity) :: exogenousAction;
    % [NOTE: 'turnto' action and 'facing' fluent are MISSING]

    loc(object) :: inertialFluent(location);
    read_or_lookat_executed :: inertialFluent(boolean).

% Generated constraints
walk(M, L) causes loc(M)=L.
% ... [Omitted standard walk/grab rules] ...

% 9. Reading causes execution flag
read_action(M, RS) causes read_or_lookat_executed.
% 10. Reading requires holding
nonexecutable read_action(M, RS) if holds_rh(M)\=RS.

% 12. Looking requires same location
nonexecutable lookat(M, RS) if loc(M)=L1 & loc(RS)=L2 & L1\=L2.

noconcurrency.
:- query
    0: ... ~read_or_lookat_executed;
    maxstep: holds_rh(character_65)=novel_1000 & read_or_lookat_executed.
```

Analysis: The model completely omits the `turnto` action and `facing` fluent. While it correctly constrains `lookat` to require spatial proximity (Constraint 12), it fails to enforce the physical requirement that the agent must face the object before looking or reading.

Affordance Violation (Task: Watch TV)

```
:- sorts
    entity >> main_entity; entity >> static_entity; location.
:- objects
    character_65 :: main_entity;
    couch_352, television_410, void :: static_entity;
    ...
:- constants
    facing(main_entity) :: inertialFluent(static_entity);
    turnto(main_entity, static_entity) :: exogenousAction;
    watch(main_entity, static_entity) :: exogenousAction;
    % [NOTE: 'lookable' property is MISSING]

% Generated constraints
turnto(M, S) causes facing(M)=S.
watch(M, S) causes lookat_watch_executed=true.

% 17. Watching requires proximity
nonexecutable watch(M, S) if loc(M)=L1 & loc(S)=L2 & L1\=L2.

% 18. Actions are not executable on 'void'.
nonexecutable watch(M, S) if S=void.

% [MISSING: nonexecutable watch(M, S) if ~lookable(S)]

noconcurrency.
:- query
    0: ... lookat_watch_executed=false;
    maxstep: ... facing(character_65)=television_410 & lookat_watch_executed=true.
```

Analysis: While this program correctly models `turnto` and location, it fails to model object affordances (e.g., `isLookable`). Constraint 18 prevents watching "void," but nothing prevents the agent from satisfying the goal by watching the `couch_352`, provided it stands next to it.

G.2. Error Analysis: GPT-4o

GPT-4o exhibited a more fundamental failure mode we term **Systemic Representational Collapse**. With only 35% of valid outputs and a task success rate of 1%, the model proved unable in face of the strict structural requirements of the $\mathcal{BC}+$ language.

Syntactic Hallucinations. The primary bottleneck was syntactic compliance. Qualitative inspection reveals that GPT-4o frequently hallucinated non-existent language features, such as embedding conditional `if` logic within `impossible` constraints (see Case 1 below) or bleeding Pythonic operators into ASP syntax. Crucially, the **Self-Revision** loop failed to rescue these cases; instead of fixing syntax, the model often degraded the program further during revisions, leading to early stops in the pipeline or severely malformed programs.

Logical Chaos. Even within the 35 cases that successfully compiled, the errors encountered with Gemini are still observed but with higher severity. Moreover, the model frequently introduced perplexing logical anomalies absent in the Gemini baseline, ranging from actions that were syntactically valid yet causally disconnected, to the construction of logical tautologies that artificially forced goal satisfaction without valid planning. In contrast to Gemini which is a stronger baseline model, GPT-4o frequently struggled with the fundamental syntactic structure of $\mathcal{BC}+$, resulting in GPT-4o's failure modes being chaotic. We demonstrate typical error cases below.

Syntax Hallucination

```
:- sorts
    entity; static_entity >> entity; ...
% ... [Standard declarations omitted] ...

% Generated constraints
sit(ME, SE) causes character_sitting = true if ME = character_65 ...
sit(character_65, couch_352) causes character_on_couch = true.

% [CRITICAL SYNTAX ERROR]
% The model attempts to use 'if' inside an 'impossible' constraint.
impossible character_sitting if ~character_on_couch.
```

```
noconcurrency.
:- query
    0: ... ~character_sitting;
    maxstep: character_sitting & character_on_couch.
```

Analysis: This is syntactic violation. The $\mathcal{BC}+$ writing instructions explicitly stated that "if" clauses are not allowed in "impossible" rules.

Revision Collapse

```
:- sorts
    main_entity; static_entity; location.
:- objects
    character_65 :: main_entity; ...
:- constants
    wash(static_entity) :: exogenousAction;
    rinse(static_entity) :: exogenousAction;
    wash_executed :: inertialFluent(boolean);
    % ...

% Generated constraints
wash(SE) causes wash_executed = true.
rinse(SE) causes rinse_executed = true.

noconcurrency.

% [CRITICAL ERROR: MISSING INITIAL STATE]
% The query block lacks the '0: ...' definition.
:- query
    maxstep: wash_executed | rinse_executed.
```

Analysis: This failure illustrates the fragility of the **Self-Revision** loop. Likely encountering an error in a previous iteration, the model attempted to rewrite the query section but accidentally truncated the initial state definition (the `0:` block). Having exhausted its revision budget, the pipeline output this incomplete, unsolvable fragment.

Severe Logical Errors

```
:- sorts
    entity >> main_entity; ...
:- constants
    % ... [Standard declarations] ...
    wash_executed :: inertialFluent(boolean);
    rinse_executed :: inertialFluent(boolean);
    % ...

% Generated constraints
grab(character_65, S) causes character_has(S).
% ... [Action rules] ...

% [CRITICAL LOGIC ERROR]
% The model forces the goal condition to be globally true using 'always'.
% This effectively "hacks" the problem, making the goal true by definition
% rather than by planning an action sequence.
always wash_executed | rinse_executed | scrub_executed | wipe_executed.

noconcurrency.
:- query
    0: grab_executed=false & wash_executed=false ...;
    maxstep: grab_executed=true & (wash_executed | rinse_executed ...).
```

Analysis: The model "hacks" the problem by accident, declaring that the cleaning tasks are already done to begin with.

Our analysis reveals that even with a highly capable foundation model like Gemini-3-Pro-Preview, the LLM+AL framework functions as a burden rather than an enhancement for embodied planning. While the pipeline demonstrates competence in simpler, logic-heavy domains where constraints and state transitions are explicit and straightforward, it struggles significantly with the tacit, unstated commonsense physics of embodied environments. The requirement to simultaneously manage strict formal syntax, causal consistency, and spatial modeling imposes a cognitive load that currently exceeds the effective window of foundation models.

H. Supplementary Experiments

This section provides additional experiments added during the rebuttal period. Including evaluation on the **BlocksWorld** benchmark, and comparison between more Neuro-Symbolic methods. Here, a **Best-of-N** setting means we consider the LLM's output correct if one out of N outputs is correct. In the **Replanning** setting, the LLM is given its last round of Input along with an error trace for the next round, and is allowed up to N iterations.

Table 14. Success Rate on the BlocksWorld Benchmark (Base Model: Qwen-Max-Latest)

Strategy	**Success Rate (%)**
Baseline	30.6%
Best-of-5	55.2%
Replanning (5 rounds)	63.4%
DecoVer (Ours)	**99.8%**

Table 15. Success Rate Comparison on Temporally Extended Cases in VirtualHome (Base Model: Qwen-Max-Latest)

Strategy	**Success Rate (%)**
Baseline	30%
Best-of-N ($N = 15$)	45%
Best-of-N ($N = 20$)	50%
Replanning ($N = 15$)	55%
Replanning ($N = 20$)	55%
DecoVer (Ours)	**70%**

Table 16. Performance comparison with additional baselines evaluated during the rebuttal phase.

Domain	**Year**	**Method**	**LLM**	**Task SR**
BlocksWorld	2026	NL-PDDL (Liu et al., 2026)	GPT-4	70.0%
	2025	PDDL-INSTRUCT (Verma et al., 2025)	GPT-4	91.0%
	-	**DecoVer (Ours)**	GPT-4	**94.4%**
	2025	LLMFP (Hao et al., 2025)	GPT-4o	96.2%
	-	**DecoVer (Ours)**	GPT-4o	**98.6%**
VirtualHome	2025	CLMASP (Lin et al., 2025)	GPT-4o	76.4%
	-	**DecoVer (Ours)**	GPT-4o	**82.9%**
	2025	D2GP (Li et al., 2025c)	GPT-4	85.8%
	-	**DecoVer (Ours)**	GPT-4	**86.2%**

We conducted a targeted ablation study on the isolated restriction generation step across two settings: w/o Objects: We omit all objects selected in Step 1. w/o Flags: We omit the flags selected in Step 3, providing only the structural flag template for the LLM to build rules from scratch. As shown in Table 17, both settings led to a marked increase in missing restrictions. These results highlight the indispensable role of explicit objects and flags during the restriction construction phase.

I. Pipeline Detail Analysis

Table 18 details the token consumption, LLM interaction frequency, and overall runtime across our benchmarks. Note that execution latency exhibits variance, particularly in **temporally extended** cases. This fluctuation primarily stems from the underlying $\mathcal{BC}+$ solver's sensitivity to the precise formulation of the generated symbolic programs. Because temporally extended tasks require exceptionally long reasoning horizons, slight non-deterministic variations in the LLM's output—such as the internal ordering of restrictions or the synthesis of functionally valid but structurally loose constraints—can drastically increase the grounding and solving time. Also, while the latency of individual LLM API queries is subject to external

Table 17. Human Evaluation of Restriction Generation across three settings.

Model	Setting	Missing Restrictions ↓
Gemini-3-Pro-Preview	**DecoVer (Ours)**	**0**
	w/o Objects	3
	w/o Flags	9
Claude-Opus-4.5	**DecoVer (Ours)**	**1**
	w/o Objects	2
	w/o Flags	13
Qwen-Max-Latest	**DecoVer (Ours)**	**3**
	w/o Objects	7
	w/o Flags	14

Table 18. Token Consumption and Execution Latency

Benchmark	Avg. LLM Calls	Avg. Tokens / Task	Avg. Tokens / Call	Runtime (s)
VirtualHome	19 (Median)	22,916	900 – 1,600	133 – 190
MCP	5	6,280	∼1,256	35 – 50
Hanoi	3	3,402	∼1,134	21 – 30

network conditions, the overall system runtime consistently scales in direct proportion to the number of required reasoning cycles (LLM calls).

Below we present a complete, successful run of DecoVer on a standard VirtualHome task. For steps with multi-round scoring, we provide only one piece of output for simplicity as the output is similar. **Task Goal:** Turn on the oven and place a pot on it.

I.1. Step 1: Object Relevance Scoring

The model first filters the open-world object list ($\mathcal{O}_{env}$) to identify the minimal set $\mathcal{O}_s$ required for the task. Based on the scoring, a pot and an oven is moving on to the next step while the other objects are discarded.

LLM Output: Object Scoring

Reasoning Trace:
Looking at the goals and current environment, I need to score each object based on its relevance to accomplishing the node goals and edge goals.
Node Goals Analysis:
- `oven` is CLOSED (already satisfied)
- `oven` is ON (needs to be changed from OFF)
- `oven` is PLUGGED_IN (already satisfied)

Edge Goals Analysis:
- `pot` is ON to `oven` (a pot needs to be placed on the oven)

Scoring Logic:

1. **oven (295):** Directly mentioned in node goals and edge goals. **Score: 100**
2. **pot (1000):** Directly mentioned in edge goals - needs to be placed ON the oven. **Score: 95**
3. **pot (1003):** Potential alternative for the edge goal. **Score: 90**
4. **character (65):** Necessary for execution. **Score: 50**
5. **dining_room (201):** Relevant for navigation. **Score: 20**
6. **bedroom (67):** Current location. **Score: 15**
7. **water (1002), food_vegetable (1001):** Irrelevant. **Score: $\leq$10**

Structured Output (JSON):

```
[
  {"Object_Name": "oven", "ID": "295", "Scoring": 100},
  {"Object_Name": "pot", "ID": "1000", "Scoring": 95},
  {"Object_Name": "pot", "ID": "1003", "Scoring": 90},
  {"Object_Name": "character", "ID": "65", "Scoring": 50},
  {"Object_Name": "dining_room", "ID": "201", "Scoring": 20},
  {"Object_Name": "bedroom", "ID": "67", "Scoring": 15},
  {"Object_Name": "water", "ID": "1002", "Scoring": 5},
  {"Object_Name": "food_vegetable", "ID": "1001", "Scoring": 3}
]
```

I.2. Step 2: Action Relevance Scoring

Based on the selected objects $\mathcal{O}_s$ and the problem description, the necessary actions are selected.

LLM Output: Action Scoring

Reasoning Trace:
Looking at the task, I need to analyze the goals and score each action accordingly.
Goal Analysis:
- **Node Goals:** Oven needs `SWITCHON` (currently OFF). Status CLOSED/PLUGGED_IN are already satisfied.
- **Edge Goals:** Pot needs `PUTON` oven. This implicitly requires `GRAB` first.
- **Current State:** Character is in bedroom; oven is elsewhere. Navigation (`WALK`) is required.

Scoring Logic:

1. **SWITCHON (95):** Directly achieves node goal (turn oven on).
2. **PUTON (95):** Directly achieves edge goal (place pot on oven).
3. **GRAB (90):** Necessary precondition for placing the pot.
4. **WALK (75):** Environmental precondition; character must reach objects.
5. **TURNTO (65):** Facilitates object interaction.
6. **Other Actions:** Low scores assigned to unrelated physical operators (e.g., EAT, CUT).

Structured Output (JSON) [Truncated for brevity]:

```
[
  {"Action": "SWITCHON", "Score": 95},
  {"Action": "PUTON", "Score": 95},
  {"Action": "GRAB", "Score": 90},
  {"Action": "WALK", "Score": 75},
  {"Action": "TURNTO", "Score": 65},
  {"Action": "RUN", "Score": 40},
  {"Action": "CLOSE", "Score": 30},
  {"Action": "OPEN", "Score": 25},
  {"Action": "PLUGIN", "Score": 20},
  ...
]
```

I.3. Step 3: Action Verification and Refinement

The LLM attempts to satisfy all goals using only the selected actions. In this case, the selected actions are enough to form a valid plan, so we proceed with actions **WALK**, **SWITCHON**, **GRAB**, **PUTON**.

LLM Output: Action Verification

Reasoning Trace:
I need to analyze whether the chosen actions are sufficient to complete all the goals.
Goal & State Gap Analysis:
- **Initial State:** Character in bedroom; Oven is OFF.
- **Goal State:** Oven ON; Pot ON Oven.
- **Action Gaps:** Need to navigate, acquire the pot, place it, and activate the oven.

Logical Sequence Simulation:

1. `WALK` to location (Required).
2. `GRAB` the pot (Required).
3. `PUTON` the pot onto the oven (Required).
4. `SWITCHON` the oven (Required).

"Wait - I need to check if there's a pot in the environment... The 'Objects_Chosen' shows a pot was selected. The actions appear sufficient. Although multiple locations might be involved, the generic `WALK` operator covers navigation."
Conclusion: The sequence `WALK` → `GRAB` → `PUTON` → `SWITCHON` is valid and covers all goals.

Structured Output (JSON):

```
[]
```

(Empty list indicates no missing actions were found; the current set is sufficient.)

I.4. Step 4: Logical Signature Construction (Part 1: Goal-Based Selection)

In this phase, the model scores the global flag library $\mathcal{F}$ based strictly on the task goals $\mathcal{G}$ and object properties.

LLM Output: Goal-Based Flag Scoring

Reasoning Trace:
Let me analyze each flag's relevance to the goals:
Node Goals:
- oven is CLOSED
- oven is ON
- oven is PLUGGED_IN

Edge Goals:
- pot is ON to oven

Action Goals:
- There is no action requirement.

Now I'll score each flag:

Structured Output (JSON):

```
[
  {"Flag": "character_location", "Score": 45},
  {"Flag": "character_facing", "Score": 30},
  {"Flag": "character_is_lying", "Score": 5},
  {"Flag": "character_is_sitting", "Score": 5},
  {"Flag": "object_is_sittable", "Score": 5},
  {"Flag": "object_is_drinkable", "Score": 5},
  {"Flag": "object_is_lookable", "Score": 5},
  {"Flag": "object_is_grabbable", "Score": 65},
  {"Flag": "object_is_eatable", "Score": 5},
  {"Flag": "object_is_lieable", "Score": 5},
  {"Flag": "object_is_typeable", "Score": 5},
  {"Flag": "object_is_readable", "Score": 5},
  {"Flag": "object_is_openable", "Score": 62},
  {"Flag": "object_is_switchable", "Score": 70},
  {"Flag": "object_is_plugable", "Score": 68},
  {"Flag": "object_location", "Score": 55},
```

```
  {"Flag": "object_is_switched_on", "Score": 95},
  {"Flag": "object_is_plugged_in", "Score": 92},
  {"Flag": "object_is_open", "Score": 88},
  {"Flag": "grabbed_item_count", "Score": 50},
  {"Flag": "object_is_inside", "Score": 25},
  {"Flag": "character_is_inside", "Score": 40},
  {"Flag": "character_is_onto", "Score": 5},
  {"Flag": "object_is_onto", "Score": 90},
  {"Flag": "object_is_grabbed", "Score": 58},
]
```

I.5. Step 5: Logical Signature Construction (Part 2: Action-Based Selection)

To ensure the signature is executable, the model re-evaluates the flag library based on the selected actions $\mathcal{A}_s$, recovering transient states required for preconditions and effects.

LLM Output: Action-Based Flag Scoring

Reasoning Trace:
Action-Flag Dependency Analysis:

- **PUTON Action:**
  - `object_is_onto` (75): Models the effect (placing object).
  - `object_is_grabbed` (90): **Critical Restriction** - object must be held before placement.
- **SWITCHON Action:**
  - `object_is_switchable` (80): Restriction - object must have a switch.
  - `object_is_switched_on` (85): Models the effect.
- **GRAB Action:**
  - `object_is_grabbable` (75): Restriction.
  - `grabbed_item_count` (70): Models capacity constraints.
- **WALK Action:**
  - `character_is_inside` (65): Models navigation effects.

Scoring Logic: Flags strictly required for the preconditions or effects of the chosen operators (e.g., `object_is_grabbed`) are boosted to ensure physical solvability, regardless of their goal relevance.

Structured Output (JSON) [Refined Selection]:

```
[
  {"Flag": "object_is_grabbed", "Score": 90},
  {"Flag": "object_is_switched_on", "Score": 85},
  {"Flag": "object_is_switchable", "Score": 80},
  {"Flag": "object_is_onto", "Score": 75},
  {"Flag": "object_is_grabbable", "Score": 75},
  {"Flag": "grabbed_item_count", "Score": 70},
  {"Flag": "grab_executed", "Score": 70},
  {"Flag": "character_is_inside", "Score": 65},
  {"Flag": "character_facing", "Score": 55},
  {"Flag": "object_location", "Score": 55},
  {"Flag": "character_location", "Score": 45},
  ...
]
```

I.6. Step 6: Causal Dependency Mapping (Flag Grouping)

Before generating ground rules, the model groups the selected flags $\mathcal{F}_s$ by the actions $\mathcal{A}_s$ that modify them. This establishes the *scope* of effect for each action. Notably, redundant flags are not included in any group, reducing hallucinated rules from the source.

LLM Output: Flag Grouping

Reasoning Trace:
Let me analyze each action and determine which flags it can modify:
Effect Analysis:
- **GRAB:**
  - `object_is_grabbed`: Becomes true.
  - `grabbed_item_count`: Increases (+1).
- **PUTON:**
  - `object_is_grabbed`: Becomes false (released).
  - `object_is_onto`: Becomes true (placed on surface).
  - `grabbed_item_count`: Decreases (-1).
- **SWITCHON:**
  - `object_is_switched_on`: Becomes true.

Unmodified Flags: Flags like `object_is_open` and `object_is_plugged_in` are present in the signature but not modified by the selected action subset (no `OPEN` or `PLUGIN` actions were chosen), so they are excluded from the modification groups.

Structured Output (JSON):

```
[
  {
    "Action": "GRAB",
    "Grouping": ["object_is_grabbed", "grabbed_item_count"]
  },
  {
    "Action": "PUTON",
    "Grouping": ["object_is_grabbed", "object_is_onto", "grabbed_item_count"]
  },
  {
    "Action": "SWITCHON",
    "Grouping": ["object_is_switched_on"]
  }
]
```

I.7. Step 7: Ground Rule Generation & Verification

In this phase, the model synthesizes the formal ground rules($\mathcal{R}_c$) based on the established dependencies. A subsequent verification step checks for logical consistency and syntax errors.

LLM Output: Rule Synthesis

Reasoning Trace:
Looking at the provided actions and flags, I'll analyze each action's effect on the associated flags.
Effect Analysis:
- **GRAB(SE):**
  - `object_is_grabbed`: Becomes true.
  - `grabbed_item_count`: Increments by 1 (arithmetic update).
- **PUTON(SE, SE1):**
  - `object_is_grabbed`: Becomes false (released).
  - `object_is_onto`: Updates to target SE1 (relational update).
  - `grabbed_item_count`: Decrements by 1.
- **SWITCHON(SE):**
  - `object_is_switched_on`: Becomes true.

Generated Rules (JSON):

```
[
  {"Action": "GRAB(SE)", "Causes": "object_is_grabbed[SE]=true"},
  {"Action": "GRAB(SE)", "Causes": "grabbed_item_count[none]+=1"},
  {"Action": "PUTON(SE,SE1)", "Causes": "object_is_grabbed[SE]=false"},
```

```
  {"Action": "PUTON(SE,SE1)", "Causes": "object_is_onto[SE]=SE1"},
  {"Action": "PUTON(SE,SE1)", "Causes": "grabbed_item_count[none]-=1"},
  {"Action": "SWITCHON(SE)", "Causes": "object_is_switched_on[SE]=true"}
]
```

Verifier Output (Self-Correction)

The model reviews the generated rules against the physical constraints of the simulation.
Verification Log:

```
[
  {
    "Rule": "GRAB(SE) causes object_is_grabbed[SE]=true",
    "Verdict": "OK",
    "Comment": "Logically sound and correctly formatted."
  },
  {
    "Rule": "GRAB(SE) causes grabbed_item_count[none]+=1",
    "Verdict": "OK",
    "Comment": "Correctly formatted with integer arithmetic."
  },
  {
    "Rule": "PUTON(SE,SE1) causes object_is_onto[SE]=SE1",
    "Verdict": "OK",
    "Comment": "Correctly formatted with static_entity value type."
  },
  ... (All rules passed validation)
]
```

Status: Verification Passed (Round 1).

Final Program Output: The verified rules, including the navigation rule for `WALK`, are compiled into the final $\mathcal{BC}+$ program.

```
{
  'Action': 'GRAB(SE)', 'Causes': 'object_is_grabbed[SE]=true',
  'Action': 'GRAB(SE)', 'Causes': 'grabbed_item_count+=1',
  'Action': 'PUTON(SE,SE1)', 'Causes': 'object_is_grabbed[SE]=false',
  'Action': 'PUTON(SE,SE1)', 'Causes': 'object_is_onto[SE]=SE1',
  'Action': 'PUTON(SE,SE1)', 'Causes': 'grabbed_item_count-=1',
  'Action': 'SWITCHON(SE)', 'Causes': 'object_is_switched_on[SE]=true',
  'Action': 'WALK(SE)', 'Causes': 'character_location=object_location[SE]'
}
```

I.8. Step 8: Restriction Generation ($\mathcal{R}_r$)

The model generates restrictions in two phases: first identifying direct state prerequisites, then detecting complex relational barriers.

LLM Output: Restriction Synthesis

Phase 1: State-Based Restrictions
Reasoning Trace:
- **PUTON(SE, SE1):** You cannot place an object unless you are holding it.
- **SWITCHON(SE):**
  - Device must be plugged in (power source required).
  - Device should not already be ON (avoid redundancy).

Generated Restrictions (JSON):

```
[
  {"Action": "PUTON(SE,SE1)", "Formula": "object_is_grabbed[SE]=false"},
  {"Action": "SWITCHON(SE)", "Formula": "object_is_plugged_in[SE]=false"},
  {"Action": "SWITCHON(SE)", "Formula": "object_is_switched_on[SE]=true"}
]
```

Phase 2: Relational Restrictions (Container Logic)
Reasoning Trace: For `GRAB(SE)`, I must check accessibility. If object SE is inside/onto container SE1, and SE1 is CLOSED, then SE is physically inaccessible.
Generated Restrictions (JSON):

```
[
  {
    "Action": "GRAB(SE)",
    "Formula": "object_is_onto[SE]=SE1 & object_is_open[SE1]=false"
  }
]
```

I.9. Step 9: Plan Generation (Symbolic Solver)

Finally, the synthesized signature, ground rules and restrictions are compiled into a $\mathcal{BC}+$ program and passed to the answer set solver (`clingo`).

Compiled $\mathcal{BC}+$ Program

```
:- maxAdditive :: 3.

:- sorts
    entity >> main_entity, static_entity;
    location.

:- objects
    character :: main_entity;
    void, oven_295, pot_1000, pot_1003 :: static_entity;
    loc_oven_295, loc_pot_1000, loc_pot_1003, start_point :: location.

:- variables
    E, E1, E2, E3, E4 :: entity;
    ME, ME1, ME2, ME3, ME4 :: main_entity;
    se,se1,se2,se3,se4, SE, SE1, SE2, SE3, SE4 :: static_entity;
    loc,loc1,loc2,loc3,loc4,Loc, Loc1, Loc2, Loc3, Loc4 :: location;
    Bool, Bool1, Bool2, Bool3, Bool4 :: boolean.

 :- constants
    puton(static_entity,static_entity) :: exogenousAction;
    switchon(static_entity) :: exogenousAction;
    grab(static_entity) :: exogenousAction;
    walk(static_entity) :: exogenousAction;
    character_location :: inertialFluent(location);
    grabbed_item_count :: additiveFluent;
    object_is_drinkable(static_entity) :: inertialFluent(boolean);
    object_is_grabbable(static_entity) :: inertialFluent(boolean);
    object_is_grabbed(static_entity) :: inertialFluent(boolean);
    object_is_onto(static_entity) :: inertialFluent(static_entity);
    object_is_open(static_entity) :: inertialFluent(boolean);
    object_is_openable(static_entity) :: inertialFluent(boolean);
    object_is_plugable(static_entity) :: inertialFluent(boolean);
    object_is_plugged_in(static_entity) :: inertialFluent(boolean);
    object_is_switchable(static_entity) :: inertialFluent(boolean);
```

```
    object_is_switched_on(static_entity) :: inertialFluent(boolean);
    object_location(static_entity) :: inertialFluent(location).

%Ground Rules
grab(se) causes object_is_grabbed(se)=true.
grab(se) increments grabbed_item_count by 1.
puton(se,se1) causes object_is_grabbed(se)=false.
puton(se,se1) causes object_is_onto(se)=se1.
puton(se,se1) decrements grabbed_item_count by 1.
switchon(se) causes object_is_switched_on(se)=true.
walk(se) causes character_location=object_location(se).

%added constraints:
nonexecutable puton(se,se1) if (object_is_grabbed(se)=false).
nonexecutable switchon(se) if (object_is_plugged_in(se)=false).
nonexecutable switchon(se) if (object_is_switched_on(se)=true).
nonexecutable grab(se) if (object_is_onto(se)=se1 &
    object_is_open(se1)=false).
nonexecutable puton(se,se1) if character_location \= object_location(se1).
nonexecutable puton(se,se1)  if object_is_grabbed(se)\=true.
nonexecutable puton(se,se1) if grabbed_item_count<=0.
nonexecutable puton(se,se1) if (se=void | se1=void).
nonexecutable switchon(se) if character_location \= object_location(se).
nonexecutable switchon(se) if object_is_switchable(se)=false.
nonexecutable switchon(se)  if object_is_switched_on(se)=true.
nonexecutable switchon(se) if se=void.
nonexecutable grab(se) if character_location \= object_location(se).
nonexecutable grab(se) if object_is_grabbable(se)=false.
nonexecutable grab(se)  if object_is_grabbed(se)\=false.
nonexecutable grab(se) if grabbed_item_count>=2.
nonexecutable grab(se) if se=void.
nonexecutable walk(se) if se=void.

noconcurrency.

:- query
   0: object_is_onto(pot_1003)=void &
      object_is_switchable(oven_295)=true &
      object_is_grabbed(pot_1000)=false &
      object_is_open(pot_1003)=false &
      object_is_onto(oven_295)=void &
      object_is_switchable(pot_1000)=false &
      character_location=start_point &
      object_is_plugged_in(oven_295)=true &
      object_is_grabbable(pot_1003)=true &
      object_is_onto(pot_1000)=void &
      grabbed_item_count=0 &
      object_is_switched_on(oven_295)=false &
      object_is_grabbed(pot_1003)=false &
      object_is_open(pot_1000)=false &
      object_location(pot_1000)=loc_pot_1000 &
      object_is_grabbed(oven_295)=false &
      object_is_grabbable(pot_1000)=true &
      object_location(pot_1003)=loc_pot_1003 &
      object_is_open(oven_295)=false &
      object_is_grabbable(oven_295)=false &
      object_location(oven_295)=loc_oven_295 &
      object_is_switchable(pot_1003)=false;
    maxstep: object_is_open(oven_295)=false &
             object_is_switched_on(oven_295)=true &
             object_is_plugged_in(oven_295)=true &
             object_is_onto(pot_1003)=oven_295.
```

Solver Output

```
/root/.cplus2asp.config: cannot open file
% Pragma: Running Query '0'.
% Pragma: Minimum Step = '0', Maximum Step = '25'.
% Pragma: Searching for 1 solutions.
clingo version 5.8.0
Solving...
Solution: 1 (Time: 0.494s)
    0:
    ACTIONS:  walk(pot_1003)
    1:
    ACTIONS:  grab(pot_1003)
    2:
    ACTIONS:  walk(oven_295)
    3:
    ACTIONS:  switchon(oven_295)
    4:
    ACTIONS:  puton(pot_1003, oven_295)
    5:
SATISFIABLE
Models       : 1+
Calls        : 6
Time         : 0.419s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.107s
```

J. Pipeline Utilities

We present all prompts used in the pipeline in this section, including prompts for generation and verification, and guidelines for certain steps. We also provide a template of our flag library. By abstracting environmental properties into high-level syntactic patterns, our system categorizes world knowledge into three fundamental dimensions: states, relations, and affordances, as detailed in Table 19). Furthermore, the structural regularity of these templates naturally facilitates automated scaling. The instantiation of specific vocabulary does not require manual authoring from scratch. Instead, an automated pipeline can be constructed using Multimodal Large Language Models to parse visual observations or environment APIs, generating candidate flags that strictly adhere to our syntactic constraints.

Table 19. Structural Templates for the DecoVer Flag Library

Cognitive Category	Structural Template Formats
Entity States	`[object]_is_[state]` `[character]_is_[state]`
Spatial Relations	`[object]_is_[relation]_[object]` `[object]_is_[relation]_[character]`
Affordances & Orientation	`[object]_is_[action]able` `[character]_is_facing_[object]`

Prompt 1: Object Scoring Prompt

```
 [Environment appended here]
As for STEP ONE, you are required to perform filtering on the input objects.
To do so, you are to score all items based on the scale of relevance of this object
to the accomplishment of the goals, Where 0 means absolutely not related and 100
means the opposite.
Relevance can be scored based on the following criteria:
1:  Is this object included in the node goals?  For example, if the node goal
requires the air_conditioner to be ON, then the air_conditioner should have a high
score.
2:  Is this object mentioned in the edge goals?  For example, if the node goal
requires the APPLE to be INSIDE the BASKET, then both the APPLE and the BASKET
should have high scores.
3:  If you think you should conduct an action from the action goals on this object,
this object should have a high score.
Objects satisfying criterion 1-2 should be scored 80-100.
Objects satisfying criterion 3 should be scored 60-79.
Other objects should be scored below 59.
Double-check:  You might have given some object a high score, but the object is
can't be accessed due to another object.  Relations like this can be found in the
"Edges" section.  In this case, the restraining object should also be scored 60-79.
Double-check:  You might notice that the same node goal may appear multiple times.
In that case, that means there are multiple occurrences of that object, and ALL
occurrences of that object should have a high score.
<scoring example>
Output your FINAL scoring distribution for the objects in JSON format, also, explain
your reasons for your scoring distribution.
[{{"Object_Name":  "object_name", "ID":"id", "Properties":  [prop1,prop2...],
"Scoring":scoring}}]
```

This prompt is used in the object selecting phase.

Prompt 2: Action Relevance Scoring

```
 [Environments and selected objects appended here]
As for STEP TWO, you are required to Choose your actions from the actions list.
To do so, you are to Score the actions from 0 to 100 by the following criteria:
1:  An action should have a high score if it is stated in the Action Goals.
2:  An action should have a high score if it is needed to achieve the object
relations in the edge goals.
3:  An action should have a high score if it is needed to achieve the desired states
in the node goals.
4:  An action should have a high score if it satisfies some ENVIRONMENTAL
preconditions for other actions.
5.  An action should have a high score if it achieves a certain character state,
allowing other actions.
6.  An action should have a high score if it is needed to interact with an object in
the task description.
Refer to the "Nodes" and "Edges" section for information.
Actions satisfying criteria 1,2,3 should be scored between 80~100.
Actions satisfying criteria 4,5,6 should be scored between 60~79.
The rest of the actions should be scored below 59.
...  [Actions List and Example Omitted for Brevity] ...
Output your scoring distribution for the actions in json format.  You are also asked
to output your reasons for your scoring distribution.
The output json string follows the format:
[{{"Action":  "action", "Score":  score}},{{"Action":"action","Score":  score}}}]
```

This prompt is used in the Action Selecting phase.

Prompt 3: Flag Relevance Scoring (Goal-Driven)

```
 [Environment, selected Objects and Actions appended here]
As for STEP THREE, you are asked to select STATUS FLAGS to model the environment.
To do so, ask yourself the question:  On a scale of 0-100, how related is this flag
to modelling the environment?
Status Flag List:
{status_flag_list}
Relevance can be scored based on the following criteria:
A flag should have a high score if:
1.  it is needed to mark an action goal--As in some action is executed or not.
2.  it is needed to model a state related to the node goals.
3.  it is needed to record the relation stated in the edge goals.
4.  it models a property/state of an object chosen in step One.
Double-Check:  Some flags follow the format:  someaction_executed.  These flags are
only to mark the execution of GOAL actions with no other purpose.
In other words, flags ending with _executed should ONLY have a high score if the
corresponding action is included in the action goals.
Flags matching criteria 1,2,3 should be scored 80~100.
Flags matching criteria 4 should be scored 60~80.
Otherwise, the flag should be scored below 59.
...  [Examples Omitted for Brevity] ...
Output your scoring in json format without changing the relative order of the flags
given in the list.  Also output the reason for your scoring distribution.
[{{"Flag":  "flag","Score":  score}}]
```

This prompt is used in the Goal-Based Flag Selection phase.

Prompt 4: Flag Relevance Scoring (Action-Driven)

```
You are a helpful AI modeling assistant.  You are given Numerous Actions previously
selected and a status flag list.  The task is to choose status flags from the list
to model the selected actions' restrictions and effects.
To do this, you are to score ALL flags based on the following criteria:
1.  A flag should have a high score if this flag acts as a restriction to an action.
For example:  the flag "object_is_openable" restricts the action "OPEN".  An action
usually only has 1 or 2 restricting flags so don't associate too many flags.
the flag "object_is_switchable" restricts the actions "SWITCHON" and "SWITCHOFF".
2.  A flag should have a high score if it models the consequence of an action.
For example:  the flag "object_is_grabbed" models the effect of "GRAB".
the flag "object_is_switched_on" models the effect of "SWITCHON" and "SWITCHOFF".
If a flag does not meet the above criteria, it should be scored between 0~59.
Otherwise, the flag should be scored between 60~100.
Note:
Do not make up actions or flags.  the list is all you need.
Double check to make sure all flags are in the correct scoring intervals.
You are asked to output in json format:
[{{"Flag":  "flag","Score":  score}}]
[Actions and Flags appended here]
```

This prompt is used in the Action-Based Flag Selection phase.

Prompt 5: Flag Grouping (Mechanism Isolation)

```
You are a helpful AI modelling assistant used for causal discoveries.  You will be
given a list of Actions and a list of Status Flags.  Your task will be to build flag
lists for every action, where every flag in the list of an action can be modified by
that action.
An action may:
1) cause some flags to become true,
2) cause some flags to become false,
3) change numerical or count-based flags.
You are to Consider explicit effects only, do not implicate extra effects for any
action.
If an action logically requires ending a previous state before producing a new one,
treat that state as ``modified''.
...  [Examples and Constraints Omitted for Brevity] ...
Especially on Integer Valued Flags:  Every Integer Valued Flag should be in at least
two groups:
Where at least one action INCREASES the flag value, and at least one action
DECREASES the flag value.
The size of each group should not Exceed 4 Flags.
Output your grouping in JSON format, With the Action Names as Keys and the Lists as
values.
[
{{"Action":  "action", "Grouping":  [Flag1,Flag2,Flag3...]}},
{{"Action":  "action", "Grouping":  [Flag2,Flag3,Flag4...]}}
]
Also, you are asked to output the reasoning for your grouping.
[Actions and Flags appended here]
```

This prompt is used in the flag grouping stage before generating ground rules.

Prompt 6: Ground Rule Generation (Rule Enumeration)

```
You are a helpful AI assistant that models cause and effect carefully.  In this task,
you are asked to build ground rules using some Actions and some status flags.
Ground Rules are rules that model the effect of an action, and causes states
transitioning/builds relations/breaks relations.
Introduction on parameters:
{introduction_on_parameters}
Instruction on filling in parameters/Values:
{instruction_on_args_and_values}
Some actions may cause this flag's value to INCREASE, and other actions may cause
this flag's value to DECREASE.
You are to Reason out:  Which action causes the first, and which action causes the
latter, and by what amount.
For example:  GRABBING an item increases the "GRABBED_COUNT" counter, and releasing
an item in anyway decreases the same counter.
You are now to generate Ground rules based on some Actions and some status flags.
Every action given follows numerous status flags.  You are to measure the Action's
Cause on it, and build a rule.
For example:
Given the input:  Action:  OPEN(SE). Grouping:  [object_is_open[SE]] You may build:
"Action":  "OPEN(SE)" , "Causes":  "Object_is_open[SE]=true.  Meaning Opening a
container causes it to be open.
...  [Example Omitted for Brevity] ...
Rules to follow:
1.  Understand the meanings of flags before writing a rule.  Do not imply extra
causes for any action.
2.  Every flag in the group should have a rule built for it with the corresponding
action, unless the hints section says otherwise.
3.  You are to fill in arguments when needed.  Never output rules with
"static_entity" in brackets.
4.  You are only allowed to use Actions and Flags given to you.
Output the rules you generated in json format.
[{{"Action":"action","Causes":"causes"}}]
[Grouping appended here]
```

This prompt is used in generating ground rules based on grouped actions and flags.

Prompt 7: Restriction Generation (Target Object State)

```
 [Environment, Actions, Flags appended here]
As for STEP Five, you are asked to Generate Restrictions For the Environment.
[Introduction On parameters]
{introduction_on_parameters}
[Introduction On Restrictions]
{introduction_on_restrictions_state}
[Notes on object states]
Object States may restrict the execution of certain actions.  These states include
devices being ON/OFF, OPEN/CLOSED, and etc.
You are to fully understand the meaning of each state to build restrictions
correctly.
[Notes on character states]
Character states may restrict the execution of certain actions.  Preconditions
needed to take into consideration is:
1):  Character's posture, either sitting/lying/standing.
2):  Character's facing/gazing direction, which is the direction the character's
face is pointed at.
...  [Logic Guide Omitted for Brevity] ...
[Notes on "OR" "AND"]
Sometimes, the restriction may be concatenated by multiple flags.
For example, consider the rule:"You can't read a book without grabbing it first".
This rule can be written as:  nonexecutable read(SE) if character_grabbing_left_hand!=SE
& character_grabbing_right_hand!=SE.
where "&" means the "AND" operator.
...  [Examples Omitted] ...
Output your ALL of your restrictions generated in Json Format:
[{{"Action":  "Action", "Formula":  "formula"}}]
Where:
Action is the action, with arguments filled in if there are any.
Formula is the formula you've built.
[Output Sample]
Follow the Output Sample to avoid format errors.
{{
["Action":"Action_1(SE)","Formula":"flag1[SE]=true"]
["Action":"Action_1(SE)","Formula":"flag2[SE]!=SE1"]
["Action":"Action_2(SE1,SE2)","Formula":"flag4[SE1]=SE2 & flag5[SE2]=false"]
}}
```

This prompt focuses on identifying conditions where an action is prohibited due to the intrinsic state of the target object.

Prompt 8: Restriction Generation (Environmental Restrictions)

```
 [Environment, actions, flags appended here]
As for STEP Five, you are asked to Generate Restrictions For the Environment.
[Introduction On parameters]
{introduction_on_parameters}
[Introduction On Restrictions]
{introduction_on_restrictions_env}
[Notes On Connected Structures]
Some Formulas may require connecting multiple flags.
For example:  "You Can't Grab something if it is located on a high shelf"
This rule can be written as:  nonexecutable GRAB(SE) if object_is_onto[SE]=SE1 &
is_too_high[SE1]=true.
Where, the object to interact with is SE, the High shelf is SE1.  "Is_too_high" and
"object_is_onto" restricts the action together.
[Notes on "OR" "AND"]
Sometimes, the restriction may be concatenated by multiple flags.
For example, consider the rule:"You can't read a book without grabbing it first".
This rule can be written as:  nonexecutable read(SE) if character_grabbing_left_hand!=SE
& character_grabbing_right_hand!=SE.
where "&" means the "AND" operator.
...  [Logic Guide Omitted] ...
Output your ALL of your restrictions generated in Json Format:
[{{"Action":  "Action", "Formula":  "formula"}}]
Where:
Action is the action, with arguments filled in if there are any.
Formula is the formula you've built.
[Output Sample]
You are to Follow the format strictly, to avoid massive errors.
{{
["Action":"Action_1(SE)","Formula":"flag1[SE]=true"]
["Action":"Action_2(SE)","Formula":"flag2[SE]=false | flag3[SE]=true"]
["Action":"Action_3(SE1,SE2)","Formula":"flag4[SE1]=SE2 & flag5[SE2]=false"]
}}
```

This prompt focuses on identifying restrictions imposed by external entities or spatial configurations.

Prompt 9: Action Set Verification

```
 [Environment, selected actions appended here]
You are now to verify:  Are the actions chosen ENOUGH to complete all the goals?  Or
is some action missing?
To do this, you are to reason out the problem:
Given ONLY these actions:  Is there a reasonable action sequence available that
after executed, all goals will be satisfied?
Notes:
1.  Observe the environment states carefully, you are asked to focus on the initial
object states and relations.
2.  Reason under the fact that the actions ONLY cause the effects in the description
with NO OTHER IMPLICATIONS. Do not make up effects for actions, the actions only
do what they are described to do.  Also, you are not to make assumptions about the
environment that works in your favor.
If not, you are to write several comments in json format:
[{{"Unsatisfiable_Prerequisite":"pre_requisite", "Comment":  "comment"}}]
"pre_requisite" should indicate the "PROBLEM" in the environment:  "STH" can't be
achieved.
and "Comment" should be a brief explanation of the unsatisfiable prerequisite.
Where both sections should be written in clear, bold language that is
easy-to-understand.
Here is an action list for reference, and you can search for missing actions in it
and add it to the comments.
Note:  Double Check The actions ALREADY chosen.  Do not raise a comment if the
action is already chosen.  For "False Positives" Can Damage the continuing process.
Also, if certain goal states are already satisfied in the initial environment, do
not raise a comment.  For this may cause confusion in future steps.
[Actions List]
{actions_list}
```

This prompt is used in the action verification step. It simulates a planning attempt to identify missing actions.

Reference: Argument and Variable Handling

```
When Building a rule or restriction, you are given flags/actions like so:
Action(static_entity)
Flag(static_entity)
Another_Flag(static_entity_src,static_entity_dest)
These flags all accept "Static Entities" as arguments, So when building a
rule/restriction, they should be used as:
Action(SE)
Flag(SE)
Another_Flag(SE1,SE2)
where SE,SE1,SE2 are variables.  Do not use actual object names in Arguments.  Also,
If a flag requires multiple arguments, make sure to use different variables for
different arguments.
```

This text is injected into the {`instruction_on_args_and_values`} placeholder found in Prompts 6, 7, and 8.

Reference: Parameter Syntax and Equation Format

```
Each status flag in this system is followed by one or two pairs of brackets.
Square Brackets "[]" --- defines the parameter(s), and the argument placed in it
describes who the flag belongs to.
Round Brackets "()" --- defines the value of this status flag.
For Example:
is_open[static_entity](boolean)
this flag describes whether an object is open or not.  The first pair of brackets
allows this flag to model the status of different objects.
object_location[static_entity](location)
this flag describes the location of some object.  Just like the previous example, it
can describe the location of ANY object.
if the Square Brackets Contains "none", meaning this flag models a environment level
or character-level status and does not accept any parameters:
For example:  character_facing[none](static_entity) is a flag with value
"static_entity".  And since it models the character only, it does not accept any
parameters.
To use a flag in a Rule/Restriction, you are to write in the form of an equation:
Flag_name[Param]=Value.
where Param is the corresponding parameter, and Value should be a value with its
type matching the requirement of the flag.
This is the ONLY VALID FORMAT for using flags in ground rules or restrictions.
[Correct Usages Example]:
Flag1[SE]=val
Flag2[SE]=SE1
Flag3[SE]=true
[Incorrect Usages]:
Flag1[SE](val) # Did not write flag in form of an equation
Flag2[SE]=none
Flag3[SE]=null #Using illegal values
```

This text is injected into the {`introduction_on_parameters`} placeholder found in Prompts 6, 7, and 8.

Reference: Target Object/Agent State Logic

```
Actions can not be executed whenever you want, and to whatever you want.
In other words, actions are bounded by the restriction:
nonexecutable <Action_Name> if <Formula>
The restrictions you are asked to generate focuses on Object States.  As In:
"The Action Cannot be executed if the target object/ Character is in some state"
To reason out this problem, you must Observe the environment carefully, and figure
out WHAT state of the object/character may restrict the action from being executed.
```

Usage: This text is injected into the {`introduction_on_restrictions_state`} placeholder found in Prompt 7.

Reference: Environmental/Spatial Logic

```
Actions can not be executed whenever you want, and to whatever you want.
In other words, actions are bounded by the restriction:
nonexecutable <Action_Name> if <Formula>
The restrictions you are asked to generate focuses on the Environment.  Environment
setting may cause some actions to be allowed in one scene, but restricted in
another.
For example,Take the action "GRAB" for example.  An item out in the open is clearly
grabbable, but you can't grab it if it is locked away in a closed container.
Here is the reasoning for this example:
"The Item to grab, also known as obj1, is INSIDE a container".
"The container, also known as obj2, is CLOSED".
These two statements TOGETHER, restrict the action GRAB.
Therefore, when building restrictions focusing on the environment, you must reason
out:
1) Status of the Item you wish to interact with, which is Obj1.  It is currently <at
some relation> with Obj2.
2) Status of Obj2.  This very status restricts the execution of the action.
```

Usage: This text is injected into the {`introduction_on_restrictions_env`} placeholder found in Prompt 8.