BeSimulator: A Large Language Model Powered Text-based Behavior Simulator

Jianan Wang[1,2], **Bin Li**[2*], **Jingtao Qi**[2], **Xueying Wang**[2], **Fu Li**[2], **Hanxun Li**[1,2]
[1]College of Computer Science and Technology, National University of Defense Technology
[2] Intelligent Game and Decision Lab (IGDL)
`wangjianan@nudt.edu.cn` `libin_bill@126.com`

Abstract

Traditional robot simulators focus on physical process modeling and realistic rendering, often suffering from high computational costs, inefficiencies, and limited adaptability. To handle this issue, we concentrate on behavior simulation in robotics to analyze and validate the logic behind robot behaviors, aiming to achieve preliminary evaluation before deploying resource-intensive simulators and thus enhance simulation efficiency. In this paper, we propose **BeSimulator**, a modular and novel LLM-powered framework, as an attempt towards behavior simulation in the context of text-based environments. By constructing text-based virtual environments and performing semantic-level simulation, BeSimulator can generalize across scenarios and achieve long-horizon complex simulation. Inspired by human cognition paradigm, it employs a "consider-decide-capture-transfer" four-phase simulation process, termed *Chain of Behavior Simulation (CBS)*, which excels at analyzing action feasibility and state transition. Additionally, BeSimulator incorporates code-driven reasoning to enable arithmetic operations and enhance reliability, and reflective feedback to refine simulation. Based on our manually constructed behavior-tree-based simulation benchmark, BTSIMBENCH, our experiments show a significant performance improvement in behavior simulation compared to baselines, ranging from 13.60% to 24.80%. Code and data are available at https://github.com/Dawn888888/BeSimulator.

1 Introduction

Simulation plays a pivotal role in robotics, providing a controlled platform for testing, enabling researchers to iteratively optimize robotic systems while circumventing the risks and costs associated with physical prototyping (Koenig and Howard, 2004). Conventional simulation tools, such as

*Corresponding authors

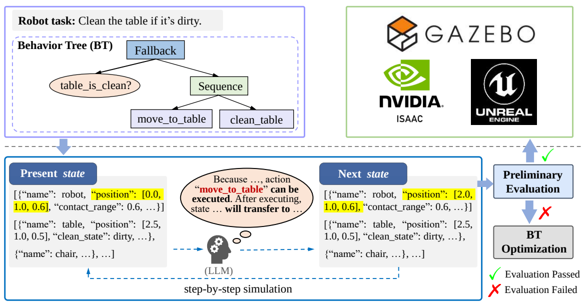

Robot task: Clean the table if it's dirty.
Behavior Tree (BT)
Fallback
table_is_clean?
Sequence
move_to_table
clean_table
GAZEBO
NVIDIA ISAAC
UNREAL ENGINE
Present state
[{"name": robot, "position": [0.0, 1.0, 0.6], "contact_range": 0.6, …}]
[{"name": table, "position": [2.5, 1.0, 0.5], "clean_state": dirty, …},
{"name": chair, …}, …]
Because …, action "move_to_table" can be executed. After executing, state … will transfer to …
(LLM)
Next state
[{"name": robot, "position": [2.0, 1.0, 0.6], "contact_range": 0.6, …}]
[{"name": table, "position": [2.5, 1.0, 0.5], "clean_state": dirty, …},
{"name": chair, …}, …]
step-by-step simulation
Preliminary Evaluation
BT Optimization
✓ Evaluation Passed
✗ Evaluation Failed

Figure 1: Workflow of BeSimulator. Based on the task description and robot behavior planning (e.g., BTs), BeSimulator employs LLMs to conduct text-based simulations for identifying behavior logic defects. This serves as a preliminary evaluation before using conventional simulators, enhancing simulation efficiency.

Gazebo (Koenig N) and Unreal Engine (Epic), have been extensively utilized for tasks involving navigation and human-robot interaction (Takaya et al., 2016; Chandan et al., 2021). However, these platforms are predominantly domain specific and focus on modeling physical processes as well as pursuing realistic rendering, which struggle with high computational demands, inefficiencies, and restricted adaptability to the dynamic and diverse nature of environments (Staranowicz and Mariottini, 2011). Additionally, they rely on domain experts to design initial simulation scenes and evaluate the results.

To alleviate this problem, we focus on behavior-level simulation, which abstracts complex physical interactions between robots and environments while maintaining action outcomes consistent with real-world scenarios. It emphasizes the control logic of robotic behavior planning solutions (BPS), such as Finite State Machine (FSM) (Lee and Yannakakis, 1996), Hierarchical Task Network (HTN) (Hayes and Scassellati, 2016), and Behavior Tree (BT) (Colledanchise and Ögren, 2018), aiming to detect the potential conflicts with reality and task logic. Compared to conventional simulators, behavior-level simulation offers greater computational efficiency and broader generalizability, enabling early detection of behavior logic defects before deploying conventional simulators, thus reducing robotic development time and cost.

World model, which originates from the mental models of humans, can predict the next *state* after executing an *action* in the present state (Ha and Schmidhuber, 2018; Matsuo et al., 2022). Trained on large-scale datasets, Large Language Models (LLMs) encapsulate rich world knowledge and have exhibited potential as sophisticated world models for reasoning and planning (Hao et al., 2023; Zhao et al., 2024). This raises a pivotal research question: can the intrinsic world modeling capacity of LLMs be utilized for robotic behavior simulation? Existing research (Wang et al., 2024) has demonstrated its feasibility in text-based environments. However, experiments show that while the direct simulation performance of LLMs is impressive, their reliability remains limited. The limitation mainly stems from two factors: the inability to capture state transitions that are indirectly related to actions and challenges in arithmetic reasoning.

To bridge this gap, we propose **BeSimulator**, a LLM-powered framework designed to efficiently simulate BPS, as an effort towards behavior simulation in text-based environments, as shown in Figure 1. BeSimulator consists of three key modules: 1) *Case Generation*, to generate the text-based simulation environment from the robot task, which includes diverse world states; 2) *BPS Simulation*, to perform step-by-step behavior simulation according to the control logic of the solution, which means conducting state transitions on the generated case states; 3) *BPS Evaluation*, to evaluate the effectiveness of the solution and think about potential defects if ineffective. In contrast to conventional simulation tools, it integrates environment design and result evaluation, thus alleviating the costly need for expert involvement. To improve the LLMs reliability, BeSimulator adopts *Chain of Behavior Simulation (CBS)*—a four-phase simulation process inspired by human cognitive reasoning paradigm—to deeply analyze the action feasibility and state transitions, especially those indirectly associated with actions. It also incorporates a code-driven reasoning mechanism to tackle arithmetic reasoning challenges. Moreover, a reflective feedback mechanism is utilized to enhance the LLMs' error recovery capability, thus refining simulation.

Given the modularity and popularity of BTs in robotic control, we construct a BT simulation benchmark, BTSIMBENCH, to evaluate our approach's performance. This benchmark provides 75 BTs based on the 25 robot tasks from BEHAVIOR-1K(Li et al., 2024a), which are long-horizon and rely on complex manipulation skills. Experimental results indicate that BeSimulator enhances the reliability of LLMs in behavior simulation, achieving higher accuracy in identifying defective behavior planning compared to the baselines.

Overall, we make the following contributions:

- We propose BeSimulator, an LLM-powered, text-based behavior simulator to efficiently analyze and validate the behavior logic of robots. Inspired by human cognition processes, it integrates the Chain of Behavior Simulation to enhance analyzing action feasibility and state transitions.
- Considering inherent biases and struggle with numeric reasoning of LLMs, BeSimulator incorporates the code-driven reasoning and reflective feedback mechanisms to enhance reliability and refine simulation.
- We construct BTSIMBENCH, a simulation benchmark based on BTs, containing various task categories and behavior types. Experimental results on BTSIMBENCH across four LLMs demonstrate BeSimulator's versatility and effectiveness in behavior simulation.

2 Related Works

2.1 Robotics Simulators

Common robot simulators include general simulators like Gazebo (Koenig N), Isaac Sim (NVIDIA), V-REP (Rohmer et al., 2013), and some game engines like Unreal Engine (Epic). Robot simulators have been widely used for tasks related to navigation (Takaya et al., 2016), human-robot interaction (Chandan et al., 2021), vehicle driving (Dosovitskiy et al., 2017), etc. Their performance is primarily determined by the physics and rendering engine. The physics engine is used to mathematically model complex physical processes like motion, collisions, etc. The rendering engine provides a visual interface to enhance simulation realism. However, these simulators suffer from low efficiency, high computational costs, poor generalization capability (Staranowicz and Mariottini, 2011; Iovino et al., 2021), and reliance on manual scene construction and expert evaluation. Based on this, we focus on behavior-level simulation to examine logical defects behind robotic behaviors, thereby achieving preliminary validation before deploying resource-intensive simulators. This can significantly reduce cycles and associated costs in robotic system development.

2.2 Conventional Behavior Simulation

Traditionally, behavioral simulation refers to simulating the behavior patterns of specific subjects. Most researches focus on user-behavior simulation to evaluate the effectiveness of evacuation systems (Kountouriotis et al., 2016; Harada et al., 2015). Moreover, some researchers focus on behavior simulation of various subjects, like infants (Nishida et al., 2004), humans living in atypical buildings (Lee, 2019), etc. There are also research efforts (Zhang et al., 2023; Hassouni et al., 2018) that utilize behavior simulators to provide a simulation environment for reinforcement learning (RL). These are fundamentally different from the behavior simulation expounded in this paper.

2.3 LLMs for Logical Reasoning

Trained on the large-scale corpus, LLMs exhibit remarkable reasoning capability. LLMs are typically prompted to decompose complex problems and engage in step-by-step thinking and reasoning, exemplified by CoT (Wei et al., 2022), Zero-shot-CoT (Kojima et al., 2022), self-consistency (Wang et al., 2022), etc. Some methods combine reasoning problems with search algorithms like ToT (Yao et al., 2024), RAP (Hao et al., 2023), etc. Moreover, some researchers focus on supervised fine-tuning of LLMs to improve reasoning, such as WOMD-Reasoning (Li et al., 2024b), CPO (Zhang et al., 2024b), etc. Methods like CoC (Li et al., 2023) and ToRA (Gou et al., 2023) apply external tools such as code interpreters, computation libraries, etc., aiming to reduce computational hallucination and enhance reasoning. Reflect (Liu et al., 2023) utilizes generated failure explanations to rectify reasoning and planning errors. In this paper, we propose Chain of Behavior Simulation containing four phases from shallow to deep, which conforms to the human cognition process further.

3 Problem Formalization

Based on research (Colledanchise and Ögren, 2018; Cai et al., 2021), a behavior planning problem can be formalized as a quintet: <$\mathcal{S}$, $\mathcal{A}$, T, $s_{initial}$, g>, where $\mathcal{S}$ is state space, $\mathcal{A}$ is action space, $T : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is the state transition rules, $s_{initial}$ is the initial scene state and g is goal condition. After executing action a in the state s_t, the next state $s_{t+1} = T(s_t, a)$. The target of the behavior planning problem is to produce a solution π capable of transferring $s_{initial}$ to g in finite steps. In this paper, we aim to determine whether one behavior planning solution is effective through behavior simulation. Our main idea is that based on the initial scene states $s_{initial}$ and the control logic of the solution π, step by step perform state transitions and finally determine whether the goal condition g is achieved, as is shown in Eq 1. In this paper, we integrate LLMs to implement this process to achieve automated and long-horizon behavior simulation.

$$g \subseteq T(s_{initial}, \pi) \tag{1}$$

4 Method

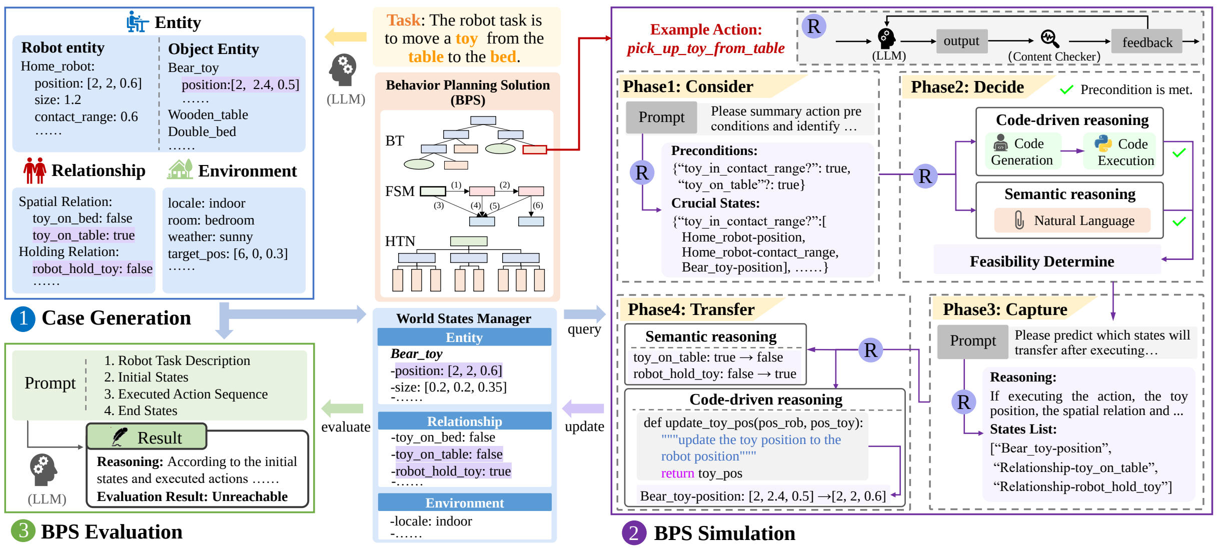
4.1 Overview

In this paper, we propose BeSimulator, illustrated in Figure 2, a modular and novel LLM-powered framework that conducts behavior simulation in text-based environments. It consists of three key modules, including *Case Generation*, *BPS Simulation*, and *BPS Evaluation*. Based on robot tasks, BeSimulator first generates simulation cases, including various world states. Then, according to the control logic of the BPS, BeSimulator analyzes the action feasibility and performs state transitions step by step. Finally, in view of the simulation results, BeSimulator evaluates whether the solution is effective.

4.2 Case Generation

Given a robot task, BeSimulator first initiates the corresponding simulation case. Leveraging the scene comprehension and generation capabilities of LLMs, the case generation exhibits strong generalization and is conducive to constructing complex scenes. The three parts of a case are as follows.

- **Entity Information** is composed of robot entities and object entities. The robot entities are autonomous and capable of action, e.g., *sweeping robots*, *quadruped robots*. The object entities are static, e.g., *tables*, *chairs*. For each entity, the generated states include basic properties: id, type, position, size, and task-related properties. E.g., if the task requires the robot to turn on lamp, the *on_off_state* property of the lamp is considered task-related.



```
def update_toy_pos(pos_rob, pos_toy):
    """update the toy position to the robot position"""
    return toy_pos
```

Figure 2: Overview of BeSimulator. **Module 1: Case Generation** generates simulation cases from task descriptions. The cases contain diverse world states, which are subsequently maintained by a world state manager. **Module 2: BPS Simulation** dynamically simulates action sequences according to the execution logic of BPS. Using single action *pick_up_toy_from_table* as an example, the schematic illustrates the four-phase "consider-decide-capture-transfer" process, which first checks the action feasibility and performs state transitions to update the manager. BeSimulator integrates code-driven reasoning and reflective feedback into the process. **Module 3: BPS Evaluation** conducts evaluation based on the simulation results.

- **Relationship Information** defines interactions between entities, including spatial relationships (e.g., *on*, *in*, *distance*), holding relationships, manipulable relationships, etc.
- **Environment Information** describes task-specific settings, e.g., *locale*, *weather*, *object target poses*.

4.3 BPS Simulation

Building upon the initial scene case, BeSimulator dynamically simulates action sequences based on the control logic of the BPS. Our approach leverages the reasoning capability and embedded knowledge of LLMs to support behavior-level simulation while significantly improving efficiency. To address the reliability limitations of direct LLM-based simulation, we propose CBS, an atomic action simulation process inspired by the human cognition processes. Additionally, we incorporate code-driven reasoning to ensure precise numerical computation and employ reflective feedback mechanism to iteratively refine simulation.

4.3.1 Chain of Behavior Simulation

For each action, BeSimulator employs chain of behavior simulation to model its execution via a structured four-phase process: consider-decide-capture-transfer. Through CBS, BeSimulator first analyzes the action's feasibility (phase 1-2) and, if feasible, predicts the resulting state transitions (phase 3-4). The prompt designs for CBS are detailed in Appendix A.

During the first two phases, BeSimulator analyzes whether the action execution adheres to the real-world logic. Specifically, in the **consider** phase, BeSimulator ponders the preconditions required for successful action execution and identifies the crucial states affecting each precondition. Subsequently, in the **decide** phase, BeSimulator assesses the satisfaction of each precondition by examining the corresponding crucial states. Then it aggregates these assessment results to determine the action's feasibility within the current scene. For instance, when simulating the action *"pick_up_toy"*, BeSimulator summarizes two preconditions: *"can_robot_touch_toy?=true"* and *"whether_robot_has_free_gripper?=true"*. For the first precondition, the crucial states inferred include *robot-position*, *toy-position* and *robot-gripper_contact_range*, which determine whether the precondition is satisfied.

Upon confirming an action's executability, BeSimulator predicts the state transitions through the last two phases. Detailedly, in the **capture** phase, it identifies which scene states will be affected by the action. Then, BeSimulator determines the precise transition rules for each impacted state in the **transfer** phase. For example, for action *"move_to_bed"*, the states to be updated may include *robot-position*, the positions of held objects, the involved spatial relationships — with the latter two are indirectly related to the action. Moreover, BeSimulator utilizes a world states manager to implement state transitions. This manager, which maintains all scene states in a structured representation and supports the state querying and updating functionalities, serves as a textual virtual environment and effectively reduces LLMs' hallucinations regarding the diverse scene states.

Compared to behavior simulation with just one phase, our four-phase reasoning paradigm decomposes the complex simulation problem through sequential, human-like cognitive processes. This significantly improves the analysis of action feasibility and state transitions (especially indirectly induced transitions), and enhances the LLMs simulation reliability, which is the core goal of our approach. We further investigate its effectiveness through ablation study in Section 5.4.1.

4.3.2 Code-driven Reasoning

While LLMs demonstrate strong capabilities in semantic reasoning, their performance often degrades when handling tasks requiring numerical reasoning. To address this limitation, BeSimulator employs a code-driven reasoning mechanism for arithmetic operations involving numerical states. This mechanism, applied during the decide and transfer phases, integrates code generation and code execution via a code interpreter to ensure computational precision and improve simulation fidelity. Taking decide phase as an example, BeSimulator dynamically selects its reasoning mode based on data types of critical states. If these states involve numerical types, it activates code-driven reasoning; otherwise, it defaults to semantic reasoning. Ablation experiment in Section 5.4.2 validates its efficacy.

4.3.3 Reflective Feedback

Required to respond in JSON format, LLMs occasionally produce errors in syntactic accuracy and semantic consistency. Therefore, BeSimulator incorporates a reflective feedback mechanism. After LLMs generate an output, an automated content checker evaluates its validity from two aspects. First, the checker performs syntactic validation by examining four key aspects: (1) adherence to JSON format, (2) completeness of JSON keys, (3) accuracy of JSON values, and (4) executability of the generated codes. Second, the checker conducts semantic validation by assessing whether the reasoning process aligns logically with the final output, identifying potential inconsistencies. If errors are detected, BeSimulator provides feedback to the LLM, guiding it to reflect and re-output. The process iterates until either all errors are resolved or a predefined feedback limit is reached. In Section 5.4.3, we demonstrate the effectiveness of the mechanism through ablation.

Category	**Reality logic**	**Task logic**
Good	✓	✓
Counterfactuals	✗	N/A
Unreachable	✓	✗

Table 1: The three categories of the evaluation results. ✓ Consistent with logic. ✗ Inconsistent with logic.

4.4 BPS Evaluation

Based on the simulation results, hSimulator classifies BPS into three distinct categories: Good, Counterfactuals, and Unreachable, as presented in Table 1. The latter two categories indicate inherent logic defects in the solution.

Specifically, during the step-by-step simulation process, if any action is infeasible—indicating conflicts between the solution's execution logic and reality logic—the solution is classified as Counterfactuals. Moreover, for solutions that maintain logical consistency with reality, BeSimulator performs rigorous evaluation based on four key elements: task objectives, initial scene states, executed action sequences, and terminal states. The evaluation yields either Good or Unreachable classification. The Good category indicates that the solution achieves the task goal at the behavior level, while Unreachable indicates fundamental incompatibility between the solution and task requirements. Consider the example of *"Clean book with rag"* task, which presents two defective solutions, namely Solution A and Solution B. Solution A, which controls the robot to perform the action *"move_to_book"* followed by *"clean_book"*, exhibits Counterfactuals because it omits the crucial step of picking up the rag before cleaning. In contrast, Solution B is classified as Unreachable because the robot only picks up the rag and the book after executing the complete solution, thereby failing to achieve the intended task goal.

5 Experiments

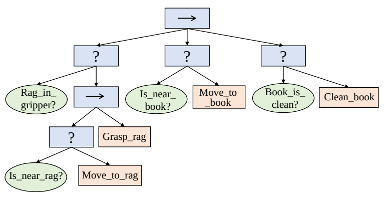
Our approach is versatile and can be adapted across different BPS. In this section, we use behavior tree (BT) as a case study and perform experiments to evaluate the effectiveness of our approach. We first propose a novel BT simulation benchmark and then conduct comprehensive experiments to address two research questions:

- To what degree of accuracy does BeSimulator conduct behavior simulation? (Section 5.3)
- To what extent do our designed mechanisms contribute to the simulation, including chain of behavior simulation, code-driven reasoning, and reflective feedback? (Section 5.4)

Node	Descriptions	Execution
Sequence	ticks its child nodes from left to right until one returns *Failure*	py_trees rules
Fallback	ticks its child nodes from left to right until one returns *Success*	
Parallel	ticks its child nodes in parallel and returns based on the setting	
Action	performs an action	CBS
Condition	checks if a condition is met	CBS

Table 2: Typical nodes in BTs and the execution rules they follow.

1. Clean pool
2. Get drink
3. Assemble furniture
4. Clean book
5. Boil water
6. Box food after lunch
7. Brew coffee
8. Chop onion



Task description: Clean a dusty book. The book is located inside a shelf initially and ……
Action nodes:
Move_to_rag: The robot moves to the location of the rag.
Clean_book: The robot uses the rag in its gripper to wipe the dust off the book and make the book clean.
……
Condition nodes:
Rag_in_gripper?: Check if the rag is in the robot's gripper.
……

Figure 3: Examples of BTSIMBENCH. Top Left: Example activities. Top right: Good BT for the *Clean book with rag* task, where blue boxes with '→' and '?' denote Sequence and Fallback nodes, respectively. Green ellipses and orange boxes represent Condition and Action nodes. Bottom: Task description and descriptions of Action and Condition nodes in the BT.

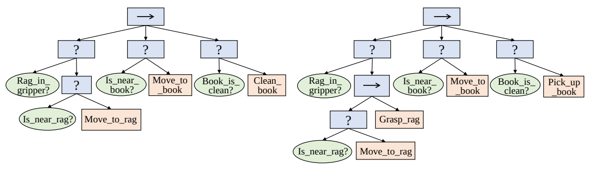


Figure 4: Bad BT examples. Left: BT of the Counterfactuals category. Right: BT of the Unreachable category.

5.1 BTs Simulation

A BT is a directed tree structure where leaf nodes (Condition and Action nodes) control the robot's perception and actions, while internal nodes (e.g., Fallback and Sequence nodes) manage the execution logic of leaf nodes (Colledanchise and Ögren, 2018), as is shown in Table 2. BT execution begins at the root node, which ticks its descendant nodes at each time step through Depth First Search (DFS). Based on the current scene, the tick creates a control flow that determines the robot's perception and action sequence.

In our experiments, we utilize py_trees[1] as BTs engineer that sends tick signal and controls nodes execution. Internal nodes follow the execution rules defined in py_trees, while leaf nodes are handled by the the CBS mechanism. Specifically, BeSimulator implements the four-phase CBS process for action node simulation and employs a two-phase variant for condition nodes. This variant adopts the first two CBS phases, "consider-decide": it first identifies relevant scene states for the condition node, then determines its output (Success/Failure) based on their values. For example, for the condition node "is_near_book?", BeSimulator infers critical states (e.g., *robot-position*, *book-position*, and *robot-contact_range*) and generates code to determine whether the node succeeds or fails.

[1]https://py-trees.readthedocs.io/en/devel/

5.2 Experimental Setup

5.2.1 Benchmarks

Based on BEHAVIOR-1K (Li et al., 2024a) which is a comprehensive benchmark for human-centered robots, we construct BTSIMBENCH, a novel BT simulation benchmark comprising 75 BTs across three categories. BEHAVIOR-1K includes the definitions of 1000 daily tasks, which are long-horizon and depend on complex manipulation skills. These tasks have been experimentally verified to be extremely challenging for current AI algorithms and are widely applied in research on behavior planning (Zhang et al., 2024a), motion control (Jiang et al., 2025), etc. We select 25 tasks from them while keeping diversity in both task categories and

Model	Method	Delivery(%)	Accuracy(%)			
			Good	CFactuals	Unreachable	Average
Claude-3.5-Sonnet	CoT	100.00	**98.40 ± 2.19**	47.20 ± 4.38	60.00 ± 2.83	68.53
	BoN	100.00	**98.40 ± 2.19**	50.40 ± 3.58	64.00 ± 2.83	70.93
	BeSimulator	100.00	88.80 ± 3.34	**97.60 ± 2.19**	**92.00 ± 2.83**	**92.80**
DeepSeek-V3	CoT	100.00	87.20 ± 1.79	53.60 ± 4.56	56.00 ± 4.00	65.60
	BoN	100.00	**88.80 ± 3.35**	52.00 ± 2.83	59.20 ± 3.35	66.67
	BeSimulator	100.00	85.60 ± 2.19	**96.80 ± 3.35**	**88.80 ± 1.79**	**90.40**
Qwen2-72B-Instruct	CoT	100.00	92.00 ± 2.83	2.40 ± 2.19	52.80 ± 4.38	49.07
	BoN	100.00	**95.20 ± 1.79**	1.60 ± 2.19	**66.40 ± 2.19**	54.40
	BeSimulator	100.00	68.80 ± 4.38	**90.40 ± 2.19**	61.60 ± 3.58	**73.60**
Llama3.1-70B-Instruct	CoT	100.00	**66.40 ± 4.56**	67.20 ± 1.79	48.80 ± 3.35	60.80
	BoN	100.00	64.40 ± 3.58	71.20 ± 1.79	32.80 ± 4.38	56.13
	BeSimulator	100.00	65.60 ± 3.58	**89.60 ± 3.58**	**68.00 ± 2.83**	**74.40**

Table 3: The comparative experiment results of BeSimulator on BTSIMBENCH across four LLMs (mean ± standard deviation from five repeated experiments).

behavior types, then we utilize LLMs to convert them from BEHAVIOR Domain Definition Language (BDDL) into textual descriptions. For each task, we first construct a Good BT and provide the corresponding node descriptions, as illustrated in Figure 3. Additionally, by altering some nodes in the Good BT, we construct a Counterfactuals BT and an Unreachable BT to simultaneously measure the simulation capability of BeSimulator for both good and bad BTs, as shown in Figure 4. For each BT, we conduct multiple rounds of manual verification and adjustment to ensure its availability. See Appendix B for more BTSIMBENCH info.

5.2.2 Baselines and Adopted LLMs

Currently there is no specific framework designed for performing behavior simulation based on LLMs. Thus, we compare our method with the scenario-agnostic methods including Chain of Thought (Wei et al., 2022) and Best of N with LLM Judge. Our method and all baselines employ few-shot examples to ensure a fair comparison.

- **Chain of Thought (CoT)**: This method takes the robot task description and the BT with node descriptions as input. Specifically, we instruct LLMs to generate intermediate reasoning steps to analyze the BT's control logic, evaluate its effectiveness, and identify potential reasons if ineffective.
- **Best of N with LLM Judge (BoN)**: According to the CoT method, we sample three candidate answers and then use LLM as a judge to select the best answer based on the correctness of the answers.

To assess our method's generalization across LLMs, we select four well-known LLMs in the field of closed source and open source as our base LLMs, including Claude-3.5-Sonnet (Anthropic), DeepSeek-V3 (DeepSeek-AI, 2025), Qwen2-72B-Instruct (Yang et al., 2024) and Llama3.1-70B-Instruct (Dubey et al., 2024).

5.2.3 Metrics

To evaluate the simulation performance of BeSimulator, we use the following metrics:

- **Delivery Rate**: This metric assesses whether an LLM-based simulator can successfully deliver a simulation result within finite reflection times of LLMs. Exceeding the predefined feedback limit (5 times in our experiment settings) will result in delivery failure.
- **Accuracy**: This metric represents the proportion of BTs that are correctly evaluated for the corresponding categories. As the key evaluation criterion, it reflects the simulator's effectiveness for behavior simulation.

5.3 Simulation Performance

In our experiments, we employ BTSIMBENCH to evaluate the efficacy of BeSimulator across four LLMs. Table 3 presents the performance comparison between our method and baselines.

The results demonstrate that BeSimulator achieves significant performance improvements across all base LLMs, showing its capability for behavior-level simulation. Specifically, for DeepSeek-V3, we observe a maximum increase of 24.80% in average accuracy, corresponding to a 37.80% relative improvement over the CoT baseline. Other LLMs exhibit enhancements ranging

Method	Delivery(%)	Accuracy(%)			
		Good	CFactuals	Unreachable	Avg
CoT	100.00	87.20	53.60	56.00	65.60
Single Phase (Section 5.4.1)	100.00	80.00	80.00	76.00	78.67
w/o Code (Section 5.4.2)	100.00	64.00	92.00	76.00	77.33
w/o Feedback (Section 5.4.3)	94.67	84.00	96.00	76.00	85.33
BeSimulator	100.00	85.60	96.80	88.80	90.40

Table 4: The ablation experiment results on BTSIM-BENCH for DeepSeek-V3. "Single-phase" refers to the ablation for CBS. "w/o Code" refers to the ablation for Code-driven Reasoning. "w/o Feedback" refers to the ablation for Reflective Feedback.

from 13.60% to 24.53%, further validating the efficacy of our approach. Moreover, the results show that the BoN baseline outperforms CoT in general. This demonstrates that LLMs are better at making choices based on candidate answers than directly generating answers.

Furthermore, the baselines' strong performance on Good BTs belies a superficial understanding of robot behavior control, as it struggles to detect hidden conflicts with reality and task logic. For instance, in the Counterfactuals BT shown in Figure 4, the baselines often fail to recognize that one precondition for the "*clean_book*" action is "*hold_rag?=true*". Consequently, it tends to classify BTs as Good, resulting in degraded performance on faulty BTs. In contrast, our method outperforms the baselines in the Counterfactuals and Unreachable categories. For instance, on the Qwen2-72B-Instruct model, our method achieves 90.40% accuracy in simulating and evaluating Counterfactuals BTs, while the BoN baseline yields 1.60% accuracy. These results indicate that BeSimulator effectively improves action execution analysis and identifies potential defects, thus facilitating iterative optimization of robot systems. Additional evaluation results for BeSimulator are provided in Appendix C.

5.4 Ablation Study

In the ablation experiments, we choose DeepSeek-V3 as the base LLM. Subsequently, we systematically remove each mechanism from our method, enabling us to pinpoint and comprehend the specific efficacy of each mechanism. The ablation experiment results are detailed in Table 4.

5.4.1 Effectiveness of Chain of Behavior Simulation

To implement the ablation analysis on the thought mode of behavior simulation, we compare the "consider-decide-capture-transfer" four-phase mode of CBS with the single-phase mode, which prompts LLMs to analyze the action feasibility and state transitions in only a single phase. We observe that, despite using the single-phase thought mode, the average accuracy across three categories increases by 13.07% compared to the CoT baseline. This confirms that the approach, which constructs the text-based simulation environment and performs state transitions according to the control logic of BPS, can effectively simulate BPS and enhance evaluation accuracy. However, our analysis of LLM outputs reveals that the single-phase thought mode, constrained by the problem's complexity, fails to sufficiently analyze action feasibility and effects, which explains its inferior performance compared to BeSimulator. This underscores the importance of CBS in enhancing action feasibility analysis and capturing state transitions, particularly those indirectly connected to actions, which improves LLMs' reliability in behavior simulation.

5.4.2 Effectiveness of Code-driven Reasoning

We conduct the ablation analysis on the effects of code-driven reasoning. We remove the code-driven reasoning in CBS, transforming it into semantic reasoning. The result indicates that three category accuracy rates decrease, particularly for the Good and Unreachable categories. We find that, in the absence of code generation and execution, LLMs consistently struggle with arithmetic calculations and comparisons. For instance, LLMs are stuck in comparing the numerical values of 1.414 and 1.0. This highlights the efficacy of the code-driven reasoning mechanism in addressing the numerical hallucination of LLMs. Furthermore, compared to the single-phase mode, "w/o Code" performs excellently in the Counterfactuals category due to the human-like thought paradigm of CBS.

5.4.3 Effectiveness of Reflective Feedback

The results of the ablation experiment on reflective feedback reveal a significant 5.33% decline in delivery rate when reflective feedback is removed. The result shows that LLMs face challenges in providing outputs that satisfy syntax requirements and semantic consistency in one response. This phenomenon substantiates the significance of reflective feedback, which narrows the gap between LLMs and idea outputs and refines simulation.

6 Conclusion

We formalize the simulation problems for behavior planning solutions to evolve the real-world simulation challenges. To enhance the efficiency and generalization of simulation, we focus on behavior simulation in robotics and propose a novel LLM-based framework, BeSimulator, as an effort toward behavior simulation in the context of text-based environments. BeSimulator first generates text-based simulation scenes, then performs semantic-level simulation and ultimately evaluates. We integrate mechanisms including Chain of Behavior Simulation, code-driven reasoning, and reflective feedback to ensure the effectiveness of the simulation. Experimental results across four LLMs on our proposed BTs simulation benchmark BTSIMBENCH demonstrate that BeSimulator achieves significant improvements in the simulation performance for long-horizon tasks while enhancing efficiency.

7 Limitations

To prove the efficacy of our work, we perform adequate experiments based on BTs. We adopt BTs due to their popularity as a robot control architecture in recent years. Although our results substantiate the effectiveness and efficiency of the proposed approach on BTs simulation, the performance has not been evaluated on other robot control architectures, such as FSMs, HTNs. An important direction for future research is to extend the application of this work to a broader range of robot control architectures.

In this work, we propose BeSimulator, a behavior-level simulator in the context of text-based environments, which has a different scope from the existing physics-based and visual simulation tools. These tools are still essential for achieving high-fidelity physical simulations, as well as in scenarios where visual information is crucial. While the primary goal of this work is to realize efficient and versatile simulation. By leveraging this work, robotic system developers can conduct preliminary evaluations and optimizations prior to employing computationally intensive, resource-demanding, and costly conventional simulation tools, thereby significantly reducing development costs and expediting the overall robotics development cycle.

8 Ethics Statement

We recognize and ensure that our study aligns with the established Code of Ethics. The focus of this article is on a novel LLM-powered framework that conducts behavior simulation in text-based environments. However, we acknowledge that as an LLM application, it may be exploited by malicious individuals. For example, if someone uses our work to simulate criminal methods, ethical concerns will arise. Therefore, we urge that such applications undergo security checks on user instructions when put into use, to identify malicious attempts.

References

Anthropic. Claude 3.5 sonnet. [Online]. https://www.anthropic.com/news/claude-3-5-sonnet.

Zhongxuan Cai, Minglong Li, Wanrong Huang, and Wenjing Yang. 2021. Bt expansion: a sound and complete algorithm for behavior planning of intelligent robots with behavior trees. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6058–6065.

Kishan Chandan, Vidisha Kudalkar, Xiang Li, and Shiqi Zhang. 2021. Arroch: Augmented reality for robots collaborating with a human. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3787–3793. IEEE.

Michele Colledanchise and Petter Ögren. 2018. *Behavior trees in robotics and AI: An introduction*. CRC Press.

DeepSeek-AI. 2025. Introducing deepseek-v3. *https://api-docs.deepseek.com/news/news1226*.

Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. Carla: An open urban driving simulator. In *Conference on robot learning*, pages 1–16. PMLR.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Epic. Unreal engine. [Online]. https://www.unrealengine.com/zh-CN/features.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Minlie Huang, Nan Duan, Weizhu Chen, et al. 2023. Tora: A tool-integrated reasoning agent for mathematical problem solving. *arXiv preprint arXiv:2309.17452*.

David Ha and Jürgen Schmidhuber. 2018. Recurrent world models facilitate policy evolution. *Advances in neural information processing systems*, 31.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

Eiji Harada, Hitoshi Gotoh, and Noorhazlinda Binti Abd Rahman. 2015. A switching action model for dem-based multi-agent crowded behavior simulator. *Safety science*, 79:105–115.

Ali el Hassouni, Mark Hoogendoorn, and Vesa Muhonen. 2018. Using generative adversarial networks to develop a realistic human behavior simulator. In *PRIMA 2018: Principles and Practice of Multi-Agent Systems: 21st International Conference, Tokyo, Japan, October 29-November 2, 2018, Proceedings 21*, pages 476–483. Springer.

Bradley Hayes and Brian Scassellati. 2016. Autonomously constructing hierarchical task networks for planning and human-robot collaboration. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5469–5476. IEEE.

Matteo Iovino, Jonathan Styrud, Pietro Falco, and Christian Smith. 2021. Learning behavior trees with genetic programming in unpredictable environments. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4591–4597. IEEE.

Yunfan Jiang, Ruohan Zhang, Josiah Wong, Chen Wang, Yanjie Ze, Hang Yin, Cem Gokmen, Shuran Song, Jiajun Wu, and Li Fei-Fei. 2025. Behavior robot suite: Streamlining real-world whole-body manipulation for everyday household activities. *arXiv preprint arXiv:2503.05652*.

Nathan Koenig and Andrew Howard. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ international conference on intelligent robots and systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. Ieee.

Howard A Koenig N. Gazebo-3d multiple robot simulator with dynamics. [Online]. https://playerstage.sourceforge.net/gazebo/gazebo.html.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Vassilios I Kountouriotis, Manolis Paterakis, and Stelios CA Thomopoulos. 2016. icrowd: agent-based behavior modeling and crowd simulator. In *Signal Processing, Sensor/Information Fusion, and Target Recognition XXV*, volume 9842, pages 259–269. SPIE.

David Lee and Mihalis Yannakakis. 1996. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123.

Yun Gil Lee. 2019. Actoviz: a human behavior simulator for the evaluation of the dwelling performance of an atypical architectural space. In *HCI International 2019-Posters: 21st International Conference, HCII 2019, Orlando, FL, USA, July 26–31, 2019, Proceedings, Part III 21*, pages 361–365. Springer.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. 2023. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474*.

Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Wensi Ai, Benjamin Martinez, et al. 2024a. Behavior-1k: A human-centered, embodied ai benchmark with 1,000 everyday activities and realistic simulation. *arXiv preprint arXiv:2403.09227*.

Yiheng Li, Chongjian Ge, Chenran Li, Chenfeng Xu, Masayoshi Tomizuka, Chen Tang, Mingyu Ding, and Wei Zhan. 2024b. Womd-reasoning: A large-scale language dataset for interaction and driving intentions reasoning. *arXiv preprint arXiv:2407.04281*.

Zeyi Liu, Arpit Bahety, and Shuran Song. 2023. Reflect: Summarizing robot experiences for failure explanation and correction. *arXiv preprint arXiv:2306.15724*.

Yutaka Matsuo, Yann LeCun, Maneesh Sahani, Doina Precup, David Silver, Masashi Sugiyama, Eiji Uchibe, and Jun Morimoto. 2022. Deep learning, reinforcement learning, and world models. *Neural Networks*, 152:267–275.

Yoshifumi Nishida, Yoichi Motomura, Koji Kitamura, and Hiroshi Mizoguchi. 2004. Infant behavior simulation based on an environmental model and a developmental behavior model. In *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No. 04CH37583)*, volume 2, pages 1555–1560. IEEE.

NVIDIA. Nvidia isaac sim. [Online]. https://developer.nvidia.com/isaac/sim.

Eric Rohmer, Surya PN Singh, and Marc Freese. 2013. V-rep: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ international conference on intelligent robots and systems*, pages 1321–1326. IEEE.

Aaron Staranowicz and Gian Luca Mariottini. 2011. A survey and comparison of commercial and open-source robotic simulator software. In *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments*, pages 1–8.

Kenta Takaya, Toshinori Asai, Valeri Kroumov, and Florentin Smarandache. 2016. Simulation environment for mobile robots testing using ros and gazebo. In *2016 20th International Conference on System*

Theory, Control and Computing (ICSTCC), pages 96–101. IEEE.

Ruoyao Wang, Graham Todd, Ziang Xiao, Xingdi Yuan, Marc-Alexandre Côté, Peter Clark, and Peter Jansen. 2024. Can language models serve as text-based world simulators? *arXiv preprint arXiv:2406.06485*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Junqi Zhang, Yiqun Liu, Jiaxin Mao, Weizhi Ma, Jiazheng Xu, Shaoping Ma, and Qi Tian. 2023. User behavior simulation for search result re-ranking. *ACM Transactions on Information Systems*, 41(1):1–35.

Xiaohan Zhang, Zainab Altaweel, Yohei Hayamizu, Yan Ding, Saeid Amiri, Hao Yang, Andy Kaminski, Chad Esselink, and Shiqi Zhang. 2024a. Dkprompt: Domain knowledge prompting vision-language models for open-world planning. *arXiv preprint arXiv:2406.17659*.

Xuan Zhang, Chao Du, Tianyu Pang, Qian Liu, Wei Gao, and Min Lin. 2024b. Chain of preference optimization: Improving chain-of-thought reasoning in llms. *arXiv preprint arXiv:2406.09136*.

Zirui Zhao, Wee Sun Lee, and David Hsu. 2024. Large language models as commonsense knowledge for large-scale task planning. *Advances in Neural Information Processing Systems*, 36.

A PROMPT

The complete prompt templates of CBS's four phases are provided in the following.

A.1 *Think* of CBS

Think Phase Prompt

```
You are a world model that can recognize and understand various
scenes in the real world well, and can determine whether the
action could be executed successfully.

### Task Description
Based on your understanding of the current world state, and
semantics of the given action, your task is to determine whether
this provided action could be executed successfully based on the
current states and action description, and gives your reason
process.

### Input Description
I will give you
(1) the current world state in dictionary (denoted as ⋆Current
States⋆).
(2) the detailed description of current world states in dictionary (
denoted as ⋆Current States Description⋆)
(3) the action description (denoted as ⋆Action⋆)

### Output Rules
1. Your output must be a dictionary. Just Three keys are included: '
thought', 'corecondition', and 'corecondition_successtag'. Please
do not output irrelevant content.
2. In the 'thought' key, you should first summarize the conditions
that need to be met and the corresponding boolean value, based
on the current world states, for the action to be executed
successfully. Then, identify which states in the current world states
 are crucial for influencing each condition. Boolean value is true,
indicating that these conditions should be met for the action
execution; Boolean value is false, indicating that these conditions
should not be met for the action execution. You should express it
as 'condition?=true' or 'condition?=false'.
3. In the 'corecondition' key, the value is a dictionary. The
dictionary includes all preconditions that affect the execution of the
 action. The keys of the dictionary should be expressed as
complete question sentences, representing each precondition.
The corresponding values should be the core states from the
current world states that are necessary to check each precondition.
 The state names should be represented as A–B–C. Keys from
different levels are connected with hyphen. Each condition
corresponds to several states in a list.
4. In the 'corecondition_successtag' key, it shows the boolean
value that each precondition in the 'corecondition' dictionary
should return for the action to be executed successfully. Ensure
that information in 'corecondition_successtag' should be consisted
with the meaning in 'thought'.
5. The response should be output in the JSON format as shown in
the example below, which should begins with ```json and ends
with ```.

⋆⋆⋆⋆⋆ Example ⋆⋆⋆⋆⋆
{SHOTS}
⋆⋆⋆⋆⋆ Example Ends ⋆⋆⋆⋆⋆

⋆Current States⋆:
{CURRENT_STATES}
⋆Current States Description⋆:
{CURRENT_STATE_DESCRIPTION}
⋆Action⋆:
{ACTION_DESCRIPTION}
⋆Output⋆:
```

A.2 *Decide* of CBS

Decide Phase Prompt (Code-driven reasoning mode)

You are a world model that can recognize and understand various scenes in the real world well, and can determine whether the action could be executed successfully.

### Task Description
Based on your understanding of the current world state, the semantics of the given action, and a condition related to whether the action can be executed. Your task is to determine whether the condition is true based on the current states and the provided core states, and gives your reason process.

### Input Description
I will give you
(1) the current world state in dictionary (denoted as ⋆Current States⋆).
(2) the detailed description of current world states in dictionary (denoted as ⋆Current States Description⋆)
(3) the action description (denoted as ⋆Action⋆)
(4) one condition related to whether the action can be executed successfully (denoted as ⋆Condition⋆).
(5) the core states which are key basis for you to determine whether this condition is true or false (denoted as ⋆Core States⋆)

### Output Rules
1. Your output must be a dictionary. Just Two keys are included: " thought" and "code". Please do not output irrelevant content.
2. In the 'thought' key, you should give your reasoning process to make the decision based on the current world states and the core states.
3. In the 'code' key, you must generate python codes to calculate, according to these values of core states. Specifically, the codes should be between with '###python' and ends with '###'. And lastly must get a boolean variable "resp" in final.
4. The response should be output in the JSON format as shown in the example below, which should begins with ```json and ends with ```.

⋆⋆⋆⋆⋆ Example ⋆⋆⋆⋆⋆
{SHOTS}
⋆⋆⋆⋆⋆ Example Ends ⋆⋆⋆⋆⋆

⋆Current States⋆:
{CURRENT_STATES}
⋆Current States Description⋆:
{CURRENT_STATE_DESCRIPTION}
⋆Action⋆:
{ACTION_DESCRIPTION}
⋆Condition⋆:
{PRECONDITION}
⋆Core States⋆:
{CORESTATES}
⋆Output⋆:

A.3 *Capture* of CBS

Capture Phase Prompt

You are a world model that can recognize and understand various scenes in the real world well, and can predict future situations.

### Task Description
Based on your understanding of the current world state, and semantics of the given action, your task is to output all states needed to be changed after the action is executed, and gives your reason process.
You need to consider the attributes of agent, objects, world environment, as well as the complex intersections of their relationships.

### Input Description
I will give you
(1) the current world state in dictionary (denoted as ⋆Current States⋆).
(2) the detailed description of current world states in dictionary (denoted as ⋆Current States Description⋆)
(3) the action description (denoted as ⋆Action⋆)

### Output Rules
1. Your output must be a dictionary. Just Two keys are included: " states_transfer" and "thought". Please do not output irrelevant content.
2. For "thought" key, please output your thought and reason process about which states needed to be changed.
3. For "states_transfer" key, its value is a list of all states from the current world states that are changed by this action, with considering of the world common knowledge and the affliation relationship provided in the current states. The state name should be represented as A–B–C. Keys from different levels are connected with short lines. Considering the dependency relationship between the states needed to be changed, these states should be output sequentially in the order in which they are updated. If there are no states needed to be changed, please output ['None'].
4. The response should be output in the standard JSON format as shown in the example below.

⋆⋆⋆⋆⋆ Example ⋆⋆⋆⋆⋆
{SHOTS}
⋆⋆⋆⋆⋆ Example Ends ⋆⋆⋆⋆⋆

⋆Current States⋆:
{CURRENT_STATES}
⋆Current States Description⋆:
{CURRENT_STATE_DESCRIPTION}
⋆Action⋆:
{ACTION_DESCRIPTION}
⋆Output⋆:

A.4 *Transfer* of CBS

Note that the *Thought* part of the following prompt is from the output of the capture phase.

Transfer Phase Prompt (Semantic reasoning mode)

```
You are a world model that can recognize and understand various
scenes in the real world well, and can predict future situations.

### Task Description
Based on your understanding of the current world state, and
semantics of the given action, your task is to simulate the
execution of the given action and predict the transition of the
designated world state after the action happens. You need to
consider the attributes of agent, objects, world environment, as
well as the complex intersections of their relationships. Consider
the immediate and potential future consequences of each action
iteratively, which must be realistic and meet real-world physical
laws.

### Input Description
I will give you
(1) the current world state (denoted as ⋆Current States⋆)
(2) the detailed description of the current world state (denoted as ⋆
Current States Description⋆)
(3) the action description (denoted as ⋆Action⋆)
(4) the complete thought about the action and its effects on the
current states (denoted as ⋆Thought⋆)
(5) the state you need to change after this action is executed (
denoted as ⋆States To Be Transferred⋆)

### Output Rules
1. You need to output the new state of '{STATE_TO_TRANSFER}'
based on the information in ⋆Thought⋆.
2. The response should be output in the standard JSON format as
shown in the example below.

⋆⋆⋆⋆⋆ Example ⋆⋆⋆⋆⋆
{SHOTS}
⋆⋆⋆⋆⋆ Example Ends ⋆⋆⋆⋆⋆

⋆Current States⋆:
{CURRENT_STATES}
⋆Current States Description⋆:
{CURRENT_STATE_DESCRIPTION}
⋆Action⋆:
{ACTION_DESCRIPTION}
⋆Thought⋆:
{THOUGHT}
⋆States to be Transferred⋆:
{STATE_TO_TRANSFER}
⋆Output⋆:
```

B BTSIMBENCH

We select two activity examples from the benchmark to demonstrate its long-horizon nature and diverse behavior types, as shown in Table 6 and 7.

C Detailed Quantitative Results

This section presents a detailed quantitative analysis of our framework, evaluating its performance on two key dimensions: (1) the accuracy of extracting action preconditions and (2) the accuracy of updating world states after action execution. The results are summarized in Table 5.

The results reveal a consistent trend across four LLMs: the framework demonstrates substantially higher performance in updating world states than in extracting action preconditions. Through in-depth analysis, we find that LLMs are prone to hallucinations about real-world physical rules and generate physically implausible statements when reasoning about preconditions. This issue manifests in three categories: missing preconditions, redundant preconditions and incorrect preconditions. For example, for the action "*open_box*", one of the preconditions should be "*is_box_inside_robot_gripper_contact?=true*", but the LLM generates the precondition "*is_box_inside_robot_gripper_contact?=false*", which does not conform to the physical rules. In the context of world state updating, we identify two main types of errors: missing state transitions and incorrect state transitions. A representative failure case is shown in Appendix D.

Model	Acc_AP(%)	Acc_WS(%)
Claude-3.5-Sonnet	94.67	98.67
DeepSeek-V3	96.00	97.33
Qwen2-72B-Instruct	78.67	94.67
Llama3.1-70B-Instruct	85.33	88.00

Table 5: The quantitative results in two dimensions. "Acc_AP" refers to the accuracy of extracting action preconditions. "Acc_WS" refers to the accuracy of updating world states.

D FAILURE CASE STUDY

Due to the inherent biases and hallucinations of LLMs, the simulation may occasionally exhibit unreliability, resulting in failure cases. Specifically, the failures primarily stem from LLMs considering redundant/irrelevant action preconditions and incorrect state transitions. This highlights the need for further improvement in the LLM's understanding of real-world physical commonsense. We select one representative case for detailed demonstration, where the DeepSeek-V3 misclassifies a "Good" BT as the "Counterfactuals" category, as illustrated in Figure 5 and Table 8.

Task Name	CleanPool
Task Desciption	The robot task is to clean a stained pool with a brush and detergent. The behavior logic of a robot should be as follows. The brush and detergent are on the floor. Robot need to use a brush and detergent to scrub the pool and then rinses the pool to make it clean. The goal is to make the pool clean. The robot has two grippers, which one gripper can hold one object at a time.
Action Nodes	**Move_to_brush**: Robot moves to the location of the brush. **Pick_up_brush**: Robot grasps and lifts the brush with it one gripper. **Move_to_detergent**: Robot moves to the location of detergent. **Pick_up_detergent**: Robot picks up the detergent with its one gripper. **Move_to_pool**: Robot moves to the location of the pool. **ApplyDetergent**: Robot applies detergent which is in its gripper to the pool surface. **ScrubPoolWithBrush**: Robot extends the gripper which holds a brush to the pool which has been applied with detergent, and uses the brush to scrub the pool. **Place_brush_detergent**: Robot releases its gripper and places the brush and detergent on the floor. **RinsePool**: Robot turns on the faucet, rinses the pool which has been applied with detergent and has been scrubbed, to make the pool clean.
Condition Nodes	**Brush_in_gripper?**: Check if the brush is in the robot's gripper. **Detergent_in_gripper?**: Check if the detergent is in the robot's gripper. **IsNearBrush?**: Check if the robot is near the brush. Require a distance less than the gripper contact range of the robot to be considered near. **IsNearDetergent?**: Check if the robot is near the detergent. Require a distance less than the gripper contact range of the robot to be considered near. **IsNearPool?**: Check if the robot is near the pool. Require a distance less than the gripper contact range of the robot to be considered near. **IsfaucetOpen?**: Check if the faucet is open.

BT (Good)	<Sequence class="SequenceNode" instance_name="clean_pool_sequence">
	  <Fallback class="FallbackNode" instance_name="hold_brush">
	    <Condition class="Brush_in_gripper?" instance_name="Brush_in_gripper?"></Condition>
	    <Sequence class="SequenceNode" instance_name="hold_brush">
	      <Fallback class="FallbackNode" instance_name="move_to_brush">
	        <Condition class="IsNearBrush?" instance_name="IsNearBrush?"></Condition>
	        <Action class="move_to_brush" instance_name="move_to_brush"></Action>
	      </Fallback>
	      <Action class="pick_up_brush" instance_name="pick_up_brush"></Action>
	    </Sequence>
	  </Fallback>
	  <Fallback class="FallbackNode" instance_name="hold_detergent">
	    <Condition class="Detergent_in_gripper?" instance_name="Detergent_in_gripper?"></Condition>
	    <Sequence class="SequenceNode" instance_name="hold_detergent">
	      <Fallback class="FallbackNode" instance_name="move_to_detergent">
	        <Condition class="IsNearDetergent?" instance_name="IsNearDetergent?"></Condition>
	        <Action class="move_to_detergent" instance_name="move_to_detergent"></Action>
	      </Fallback>
	      <Action class="pick_up_detergent" instance_name="pick_up_detergent"></Action>
	    </Sequence>
	  </Fallback>
	  <Fallback class="FallbackNode" instance_name="move_to_pool">
	    <Condition class="IsNearPool?" instance_name="IsNearPool?"></Condition>
	    <Action class="move_to_pool" instance_name="move_to_pool"></Action>
	  </Fallback>
	  <Sequence class="SequenceNode" instance_name="clean_pool_sequence">
	    <Action class="ApplyDetergent" instance_name="ApplyDetergent"></Action>
	    <Action class="ScrubPoolWithBrush" instance_name="ScrubPoolWithBrush"></Action>
	    <Action class="Place_brush_detergent" instance_name="Place_brush_detergent"></Action>
	    <Fallback class="FallbackNode" instance_name="turn_on_faucet">
	      <Condition class="IsfaucetOpen?" instance_name="IsfaucetOpen?"></Condition>
	      <Action class="RinsePool" instance_name="RinsePool"></Action>
	    </Fallback>
	  </Sequence>
	</Sequence>

BT (Counterfactuals)

```
<Sequence class="SequenceNode" instance_name="clean_pool_sequence">
  <Fallback class="FallbackNode" instance_name="hold_brush">
    <Condition class="Brush_in_gripper?" instance_name="Brush_in_gripper?"></Condition>
    <Action class="pick_up_brush" instance_name="pick_up_brush"></Action>
  </Fallback>
  <Fallback class="FallbackNode" instance_name="hold_detergent">
    <Condition class="Detergent_in_gripper?" instance_name="Detergent_in_gripper?"></Condition>
    <Action class="pick_up_detergent" instance_name="pick_up_detergent"></Action>
  </Fallback>
  <Fallback class="FallbackNode" instance_name="move_to_pool">
    <Condition class="IsNearPool?" instance_name="IsNearPool?"></Condition>
    <Action class="move_to_pool" instance_name="move_to_pool"></Action>
  </Fallback>
  <Sequence class="SequenceNode" instance_name="clean_pool_sequence">
    <Action class="ApplyDetergent" instance_name="ApplyDetergent"></Action>
    <Action class="ScrubPoolWithBrush" instance_name="ScrubPoolWithBrush"></Action>
    <Action class="Place_brush_detergent" instance_name="Place_brush_detergent"></Action>
    <Fallback class="FallbackNode" instance_name="turn_on_faucet">
      <Condition class="IsfaucetOpen?" instance_name="IsfaucetOpen?"></Condition>
      <Action class="RinsePool" instance_name="RinsePool"></Action>
    </Fallback>
  </Sequence>
</Sequence>
```

BT (Unreachable)

```
<Sequence class="SequenceNode" instance_name="clean_pool_sequence">
  <Fallback class="FallbackNode" instance_name="hold_brush">
    <Condition class="Brush_in_gripper?" instance_name="Brush_in_gripper?"></Condition>
    <Sequence class="SequenceNode" instance_name="hold_brush">
      <Fallback class="FallbackNode" instance_name="move_to_brush">
        <Condition class="IsNearBrush?" instance_name="IsNearBrush?"></Condition>
        <Action class="move_to_brush" instance_name="move_to_brush"></Action>
      </Fallback>
      <Action class="pick_up_brush" instance_name="pick_up_brush"></Action>
    </Sequence>
  </Fallback>
  <Fallback class="FallbackNode" instance_name="hold_detergent">
    <Condition class="Detergent_in_gripper?" instance_name="Detergent_in_gripper?"></Condition>
    <Sequence class="SequenceNode" instance_name="hold_detergent">
      <Fallback class="FallbackNode" instance_name="move_to_detergent">
        <Condition class="IsNearDetergent?" instance_name="IsNearDetergent?"></Condition>
        <Action class="move_to_detergent" instance_name="move_to_detergent"></Action>
      </Fallback>
      <Action class="pick_up_detergent" instance_name="pick_up_detergent"></Action>
    </Sequence>
  </Fallback>
  <Fallback class="FallbackNode" instance_name="move_to_pool">
    <Condition class="IsNearPool?" instance_name="IsNearPool?"></Condition>
    <Action class="move_to_pool" instance_name="move_to_pool"></Action>
  </Fallback>
  <Sequence class="SequenceNode" instance_name="clean_pool_sequence">
    <Action class="ApplyDetergent" instance_name="ApplyDetergent"></Action>
    <Action class="ScrubPoolWithBrush" instance_name="ScrubPoolWithBrush"></Action>
  </Sequence>
</Sequence>
```

Table 6: Example 1 *CleanPool* in BTSIMBENCH. **Fault of the Counterfactuals BT:** robot does not move near the brush and detergent before picking up them, which may be out of the robot's contact range. **Fault of the Unreachable BT:** robot does not rinse the pool in the end while the task goal is to make the pool clean.

Task Name	BrewCoffee
Task Desciption	The robot task is to brew coffee. The behavior logic of a robot should be as follows. The coffee beans are stored in a open jar on the countertop, and the water source is the sink in the kitchen.An clean and empty bottle is near the sink. A coffee machine and a mug are also on the countertop. The robot needs to use the coffee machine to brew the coffee using the coffee beans and water, and then pour the brewed coffee into the mug. The goal is to ensure that the coffee is brewed and contained in the mug. The robot has two grippers, which one gripper can hold one object at a time.
Action Nodes	**Move_to_sink**: The robot moves to the location of the sink. **Grasp_water_bottle**: The robot grasps and lifts the water bottle with its one gripper. **Fill_water_bottle**: The robot turns on the sink switch, fills the bottle which is in its gripper with the flowing water until the bottle is full, and then turns off the sink switch. **Move_to_beans_jar**: The robot moves to the location of the coffee beans jar. **Grasp_coffee_beans**: The robot grasps a sufficient amount of coffee beans from the jar with its one gripper. **Move_to_machine**: The robot moves to the location of the coffee machine. **Pour_coffee_beans**: The robot pours the coffee beans which are in its gripper into the coffee machine. **Pour_water_to_mach**: The robot pours the water from the bottle which is in its gripper into the coffee machine. **Start_coffee_brewing**: The robot turns on the coffee machine power switch. **Wait_for_coffee_brew**: The robot waits for until the coffee machine gets the brewed coffee. **Move_to_mug**: The robot moves to the location of the mug. **Grasp_mug**: The robot grasps and lifts the mug. **Pour_coffee_into_mug**: The robot pours the brewed coffee from the coffee machine into the mug.
Condition Nodes	**is_near_beans_jar?**: Check if the robot is near the coffee beans jar. Require a distance less than the gripper contact range of the robot to be considered near. **beans_in_gripper?**: Check if the coffee beans is in the robot's gripper. **is_near_sink?**: Check if the robot is near the sink. Require a distance less than the gripper contact range of the robot to be considered near. **water_in_bottle?**: Check if the bottle is filled with water. **is_near_machine?**: Check if the robot is near the coffee machine. Require a distance less than the gripper contact range of the robot to be considered near. **beans_in_machine?**: Check if the coffee beans have been poured into the coffee machine. **water_in_machine?**: Check if the water has been poured into the coffee machine. **brewing_started?**: Check if the power state of the coffee machine is on and start brewing coffee based on ingredients such as coffee beans and water. **brewing_completed?**: Check if the coffee brewing process of coffee machine has completed and the content of the coffee machine is brewed coffee. **mug_in_gripper?**: Check if the mug is in the robot's gripper. **coffee_in_mug?**: Check if the brewed coffee has been into the mug.

BT (Good)	`<Sequence class="SequenceNode" instance_name="Brew_Coffee_Complete_Mug">`
	`  <Sequence class="SequenceNode" instance_name="Prepare_Coffee_Brewing">`
	`    <Sequence class="SequenceNode" instance_name="Gather_Coffee_Ingredients">`
	`      <Fallback class="FallbackNode" instance_name="Move_to_sink">`
	`        <Condition class="is_near_sink?" instance_name="is_near_sink?"></Condition>`
	`        <Action class="Move_to_sink" instance_name="Move_to_sink"></Action>`
	`      </Fallback>`
	`      <Action class="Grasp_water_bottle" instance_name="Grasp_water_bottle"></Action>`
	`      <Fallback class="FallbackNode" instance_name="Fill_water_bottle">`
	`        <Condition class="water_in_bottle?" instance_name="water_in_bottle?"></Condition>`
	`        <Action class="Fill_water_bottle" instance_name="Fill_water_bottle"></Action>`
	`      </Fallback>`
	`      <Fallback class="FallbackNode" instance_name="Move_to_coffee_beans">`
	`        <Condition class="is_near_beans_jar?" instance_name="is_near_beans_jar?"></Condition>`
	`        <Action class="Move_to_beans_jar" instance_name="Move_to_beans_jar"></Action>`
	`      </Fallback>`
	`      <Fallback class="FallbackNode" instance_name="Grasp_coffee_beans">`
	`        <Condition class="beans_in_gripper?" instance_name="beans_in_gripper?"></Condition>`
	`        <Action class="Grasp_coffee_beans" instance_name="Grasp_coffee_beans"></Action>`
	`      </Fallback>`
	`    </Sequence>`
	`    <Sequence class="SequenceNode" instance_name="Setup_Coffee_Machine">`
	`      <Fallback class="FallbackNode" instance_name="Move_to_machine">`
	`        <Condition class="is_near_machine?" instance_name="is_near_machine?"></Condition>`
	`        <Action class="Move_to_machine" instance_name="Move_to_machine"></Action>`
	`      </Fallback>`
	...(continued on next page)

BT (Good)

```
            ...
            <Fallback class="FallbackNode" instance_name="Pour_coffee_beans">
                <Condition class="beans_in_machine?" instance_name="beans_in_machine?"></Condition>
                <Action class="Pour_coffee_beans" instance_name="Pour_coffee_beans"></Action>
            </Fallback>
            <Fallback class="FallbackNode" instance_name="Pour_water_to_mach">
                <Condition class="water_in_machine?" instance_name="water_in_machine?"></Condition>
                <Action class="Pour_water_to_mach" instance_name="Pour_water_to_mach"></Action>
            </Fallback>
        </Sequence>
    </Sequence>
    <Sequence class="SequenceNode" instance_name="Brew_Pour_Coffee_Mug">
        <Fallback class="FallbackNode" instance_name="Start_coffee_brewing">
            <Condition class="brewing_started?" instance_name="brewing_started?"></Condition>
            <Action class="Start_coffee_brewing" instance_name="Start_coffee_brewing"></Action>
        </Fallback>
        <Fallback class="FallbackNode" instance_name="Wait_for_coffee_brew">
            <Condition class="brewing_completed?" instance_name="brewing_completed?"></Condition>
            <Action class="Wait_for_coffee_brew" instance_name="Wait_for_coffee_brew"></Action>
        </Fallback>
        <Fallback class="FallbackNode" instance_name="Move_to_mug">
            <Condition class="mug_in_gripper?" instance_name="mug_in_gripper?"></Condition>
            <Sequence class="SequenceNode" instance_name="hold_mug">
                <Action class="Move_to_mug" instance_name="Move_to_mug"></Action>
                <Action class="Grasp_mug" instance_name="Grasp_mug"></Action>
                <Action class="Move_to_machine" instance_name="Move_to_machine"></Action>
            </Sequence>
        </Fallback>
        <Fallback class="FallbackNode" instance_name="Pour_coffee_into_mug">
            <Condition class="coffee_in_mug?" instance_name="coffee_in_mug?"></Condition>
            <Action class="Pour_coffee_into_mug" instance_name="Pour_coffee_into_mug"></Action>
        </Fallback>
    </Sequence>
</Sequence>
```

BT (Counterfactuals)

```
<Sequence class="SequenceNode" instance_name="Brew_Coffee_Complete_Mug">
    <Sequence class="SequenceNode" instance_name="Prepare_Coffee_Brewing">
        <Sequence class="SequenceNode" instance_name="Gather_Coffee_Ingredients">
            <Fallback class="FallbackNode" instance_name="Move_to_sink">
                <Condition class="is_near_sink?" instance_name="is_near_sink?"></Condition>
                <Action class="Move_to_sink" instance_name="Move_to_sink"></Action>
            </Fallback>
            <Action class="Grasp_water_bottle" instance_name="Grasp_water_bottle"></Action>
            <Fallback class="FallbackNode" instance_name="Fill_water_bottle">
                <Condition class="water_in_bottle?" instance_name="water_in_bottle?"></Condition>
                <Action class="Fill_water_bottle" instance_name="Fill_water_bottle"></Action>
            </Fallback>
            <Fallback class="FallbackNode" instance_name="Move_to_coffee_beans">
                <Condition class="is_near_beans_jar?" instance_name="is_near_beans_jar?"></Condition>
                <Action class="Move_to_beans_jar" instance_name="Move_to_beans_jar"></Action>
            </Fallback>
        </Sequence>
        <Sequence class="SequenceNode" instance_name="Setup_Coffee_Machine">
            ... (same to the above Good BT)
        </Sequence>
    </Sequence>
    <Sequence class="SequenceNode" instance_name="Brew_Pour_Coffee_Mug">
        ... (same to the above Good BT)
    </Sequence>
</Sequence>
```

BT (Unreachable)

```
<Sequence class="SequenceNode" instance_name="Brew_Coffee_Complete_Mug">
    <Sequence class="SequenceNode" instance_name="Prepare_Coffee_Brewing">
        ... (same to the above Good BT)
    </Sequence>
    <Sequence class="SequenceNode" instance_name="Brew_Pour_Coffee_Mug">
        <Fallback class="FallbackNode" instance_name="Start_coffee_brewing">
            <Condition class="brewing_started?" instance_name="brewing_started?"></Condition>
            ...(continued on next page)
```

BT (Unreachable)	... <Action class="Start_coffee_brewing" instance_name="Start_coffee_brewing"></Action> </Fallback> <Fallback class="FallbackNode" instance_name="Move_to_mug"> <Condition class="mug_in_gripper?" instance_name="mug_in_gripper?"></Condition> <Sequence class="SequenceNode" instance_name="hold_mug"> <Action class="Move_to_mug" instance_name="Move_to_mug"></Action> <Action class="Grasp_mug" instance_name="Grasp_mug"></Action> <Action class="Move_to_machine" instance_name="Move_to_machine"></Action> </Sequence> </Fallback> <Fallback class="FallbackNode" instance_name="Wait_for_coffee_brew"> <Condition class="brewing_completed?" instance_name="brewing_completed?"></Condition> <Action class="Wait_for_coffee_brew" instance_name="Wait_for_coffee_brew"></Action> </Fallback> </Sequence> </Sequence>

Table 7: Example 2 *BrewCoffee* in BTSIMBENCH. **Fault of the Counterfactuals BT:** robot does not get coffee beans before try to pour some into the coffee machine. **Fault of Unreachable BT:** robot does not pour the brewed coffee into the mug while the task goal is to ensure that the coffee is brewed and contained in the mug.

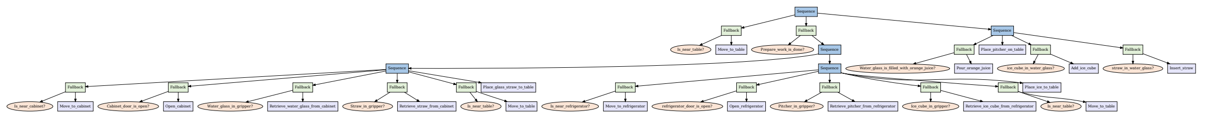


Figure 5: The BT of the failure case.

Task Name	GetDrink
Task Desciption	The robot task is to prepare a drink. The behavior logic of a robot should be as follows. The water glass and a straw are inside a cabinet. The pitcher filled with orange juice is inside an electric refrigerator. And an ice cube is inside a bowl which is also in the refrigerator. The robot needs to retrieve the water glass, pitcher, ice cube from their respective locations in the kitchen, then place them on the table. Then robot should fill the water glass with orange juice, add an ice cube to the glass, and place the straw inside the glass. The goal is to obtain a glass of orange juice with ice cube.
Action Nodes	**Move_to_table**: The robot moves to the location of the table. **Move_to_cabinet**: The robot moves to the location of the cabinet. **Open_cabinet**: The robot opens the cabinet door with one gripper. **Retrieve_water_glass_from_cabinet**: The robot extends its an gripper into the cabinet, holds the water glass and takes it out of the cabinet. **Retrieve_straw_from_cabinet**: The robot extends its an gripper into the cabinet, holds a straw and takes it out of the cabinet. **Place_glass_straw_to_table**: The robot extends the gripper that holds the water glass and the straw, places the water glass and straw on the table, and then retracts the grippers. **Move_to_refrigerator**: The robot moves to the location of the refrigerator. **Open_refrigerator**: The robot opens the refrigerator door with one gripper. **Retrieve_pitcher_from_refrigerator**: The robot extends its an gripper into the refrigerator, holds the pitcher, and takes the pitcher out of the refrigerator. **Retrieve_ice_cube_from_refrigerator**: The robot extends its an gripper into the refrigerator, holds the ice cube, and takes ice cube out of the refrigerator. **Place_ice_to_table**: The robot extends the gripper that holds ice cube, places ice cube on the table and then retracts the grippers. **Pour_orange_juice**: The robot picks up the pitcher, and then pours orange juice in the pitcher into the water glass until the glass is full. **Place_pitcher_on_table**: The robot extends the gripper that holds pitcher, places the pitcher on the table and releases the gripper. **Add_ice_cube**: The robot picks up the ice cube from the table and adds them to the water glass. **Insert_straw**: The robot picks up the straw from the table and inserts it into the water glass.
Condition Nodes	**Prepare_work_is_done?**: Check if the robot has placed ice cube, pitcher, water glass and straw on the table. **Is_near_cabinet?**: Check if the robot is near the cabinet. Require a distance less than the gripper contact range of the robot to be considered near. **Cabinet_door_is_open?**: Check if the cabinet door is open. **Water_glass_in_gripper?**: Check if the water glass is in the robot's gripper. **Straw_in_gripper?**: Check if the straw is in the robot's gripper. **Is_near_table?**: Check if the robot is near the table. Require a distance less than the gripper contact range of the robot to be considered near. **Is_near_refrigerator?**: Check if the robot is near the refrigerator. Require a distance less than the gripper contact range of the robot to be considered near. **Refrigerator_door_is_open?**: Check if the refrigerator door is opened. **Pitcher_in_gripper?**: Check if the pitcher is in the robot's gripper. **Ice_cube_in_gripper?**: Check if the ice cube are in the robot's gripper. **Water_glass_is_filled_with_orange_juice?**: Check if the water glass is filled with orange juice. **Ice_cube_in_water_glass?**: Check if the water glass contains the ice cube. **Straw_in_water_glass?**: Check if the straw is inserted into the water glass.
BT Category	Good Category
Simulation Result	Counterfactuals Category
Failure Analysis	**Failure reason:** Incorrect state transition. **Analysis:** After the robot retrieves the straw and glass and places them on the table, it retrieves ice cubes from the refrigerator. The next actions are Move_to_table and Place_ice_to_table. For the Move_to_table action, the LLM identifies three states transitions, including robot-position, ice_cube-position and relation-ice_cube_on_table. The relation ice_cube_on_table needs to transition from false to true, which is an incorrect transition and prevents the execution of the next Place_ice_to_table action. Ultimately, the behavior tree is mistakenly classified under the Counterfactuals category.

Table 8: Failure case.