2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2025)
October 19-25, 2025. Hangzhou, China

2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) | 979-8-3315-4393-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/IROS60139.2025.11245926

Look Before You Leap: Using Serialized State Machine for Language Conditioned Robotic Manipulation

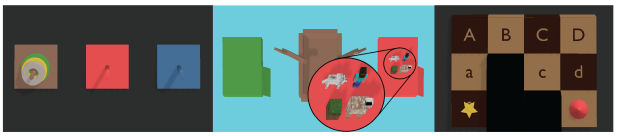
Tong Mu[1], Yihao Liu[1,2,*], Mehran Armand[1,2]

***Abstract*— Imitation learning frameworks for robotic manipulation have drawn attention in the recent development of language model grounded robotics. However, the success of the frameworks largely depends on the coverage of the demonstration cases: When the demonstration set does not include examples of how to act in all possible situations, the action may fail and can result in cascading errors. To solve this problem, we propose a framework that uses serialized Finite State Machine (FSM) to generate demonstrations and improve the success rate in manipulation tasks requiring a long sequence of precise interactions. To validate its effectiveness, we use environmentally evolving and long-horizon puzzles that require long sequential actions. Experimental results show that our approach achieves a success rate of up to 98% in these tasks, compared to the controlled condition using existing approaches, which only had a success rate of up to 60%, and, in some tasks, almost failed completely. The source code for this project can be accessed at `https://imitate.finite-state.com/`.**

I. Introduction

The recent advancements in combining Large Language Models (LLMs) with robotic systems have shown potential in automating complex task planning and execution [1]–[7]. Yet, LLMs have limited capacity for translating high-level task description text into strong, executable policies over manipulation tasks with long horizons. Existing approaches, such as those that leverage LLMs to generate task demonstration code for imitation learning [3], [4], extend on the classic philosophy of using language-conditioned policies to set goals and transfer concepts across different tasks [8]. The required training after the demonstration collection makes the framework data-dependent. These demonstrations can be human-curated [8] or LLM-generated [3], [4], and they often rely on an assumption for the task where all the objects are randomly spawned in simulation when generating a demonstration.

The current approaches work well for short-horizon tasks, where only a few sequential actions are required, or loosely constrained tasks, where precise positioning or ordering is less critical. However, they may not scale to scenarios that require precise state-dependent reasoning. For example, the reliance on vision for imitation learning [3], [8], [9] leads to failures in manipulation during the execution of long-horizon tasks. We study in particular the cascading errors when encountering previously unseen environmental structures due to limited coverage of the randomly generated data.



	Towers of Hanoi	River Crossing	Chess
Rigid Objects	3	4	2
Valid Operations	9	16	18
Valid States	27	40	90
Valid Transitions	78	92	324

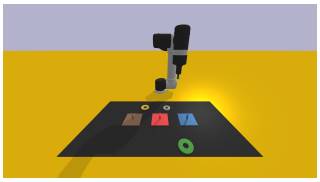
Fig. 1: Initial configurations of the designed long-horizon tasks and their finite state machine complexity. The three tasks are defined in Section III-E, and a mathematical formulation for River Crossing is defined in the Appendix.

This limitation becomes evident in our experimental findings that will be presented in the results of this work: Current frameworks are prone to cascading failure in long-horizon tasks using vision. We design three long-horizon tasks for demonstration, shown in Fig. 1. For example, in a classic task, Tower of Hanoi [3], [8], the primitive tasks are to move a ring of a certain color to a stand in another color. Existing approaches with objects spawned arbitrarily on the table are trained on randomly initialized sub-task demonstrations as shown in Fig. 2a. They may place rings on incorrect stands or miss valid pick points during the tests. This occurs because randomized training distributions fail to capture the structural consistency required for sequential spatial reasoning (e.g., stand/ring relationships depend on prior moves). In other words, when objects are subject to dynamically evolving spatial constraints, policies trained on randomly initialized demonstrations can experience performance degradation due to the divergence of the training and the evolving execution environment structures.

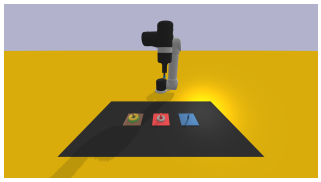
We aim to address this issue by dividing complex tasks into sub-actions, predicting possible intermediate environment structures, and training the imitation framework on all or most predicted states. These can be concluded into two key principles: "*divide and concur*" and "*look before you leap*". To achieve this, we propose a state-aware imitation learning pipeline that systematically links finite state machine (FSM) with environment-conditioned robot policy training (Fig. 3). We validate our framework on long-horizon tasks and found that it improves the success rate from up to 60% in existing approaches to almost 100%. In summary, the proposed framework has the following features that

This work was supported by National Institute of Arthritis and Musculoskeletal and Skin Diseases (R01AR080315) and National Institute of Biomedical Imaging and Bioengineering (R01EB023939).

[1] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, USA
[2] Institute for Integrative & Innovative Research and Department of Mechanical Engineering, U. of Arkansas, Fayetteville, AR 72701, USA
* Corresponding author (yliu333@jhu.edu)



(a) The scene with all randomly spawned objects.



(b) The scene of a valid state in Towers of Hanoi.

Fig. 2: Comparison between (a) a randomly initialized scene generated by the existing data collection approach and (b) a constrained, real-world scene encountered by the system during the Towers of Hanoi task.

TABLE I: Comparison of prior robot planning and imitation-learning approaches versus our FSM-driven, state-aware method.

Category	Key Idea	Limitations
Search-based Planning	Graph/tree search in predefined domain	Poor generalization to novel states
LLM-driven Planning	Prompt→plan text	Lacks explicit state feedback
Classic IL	Human demos, hard-coded rules	Data-hungry, not state-aware
Language-conditioned IL	Natural language + demonstrations	Limited to short horizons
Ours: FSM + LLM	SMSL serializes all states & demos	—

distinguish this work from previous research:

- By introducing the State Machine Serialization Language (SMSL) as a guide into demonstration generation for imitation learning, the system ensures that the resulting policy training is aware of the precise geometric constraints of complex long-horizon tasks. The encoding of symbolic states and transitions ensures that the demonstrations reflect valid and meaningful state progressions throughout the task.
- The planning process is highly deterministic because our framework uses SMSL's state transitions to propagate and record the geometric configuration of objects at each step of the task. These configurations are stored as persistent environmental states, enabling deterministic scene initialization for demonstrations.
- The predicted states are filtered to ensure correctness, and the entire demonstration generation process, including planning, filtering, and code generation, is all performed by LLM agents. These agents apply symbolic reasoning and task constraints to ensure the validity of all intermediate and goal states without relying on purely randomized data.

II. Related Works

A. Robot Task Planning

1) Search-Based and Heuristic Methods: A common solution to complex robotic tasks is to split them into easier, more manageable sub-tasks. When a robot needs to achieve a goal that cannot be done with a single action, it will have to plan the entire sequence of actions. Most of these approaches solve high-level planning through search-based methods within predefined domains [10]–[12]. Some research also focuses on heuristic strategies for robot task planning [13]–[15].

2) LLM-Driven Planning: Recent works leverage LLMs for robot task planning because LLMs can perform multi-task generalization when provided with a designed prompt input. Many recent works utilize prompt engineering for LLMs to generate text as a guide for robot task planning [16]–[19]. For long-horizon tasks, approaches generally plan in a hierarchical manner for the defined task-primitives, and common methods include trees or optimization-based algorithms [20], [21]. In [22], the authors introduce situated awareness in LLM-based robot task planning, fulfilling the missing state feedback from the environment when using LLM for task planning. To make the process more deterministic, Liu and Armand [19] proposed using LLM-generated FSMs. This aligns with our goal of addressing the problem of unseen environmental states.

B. Imitation Learning (IL) for Robotics Manipulation

1) Classic and Few-Shot IL: In traditional methods, machines and robots could only learn autonomous behaviors when experts spent time writing hard-coded rules for them. Imitation learning offers a way to teach robots by "showing" them what to do, simplifying the training of robots to perform new tasks [23]–[25]. It has demonstrated effectiveness in learning grasping and pick-and-place tasks from low-dimensional states [26], [27]. Some previous works also focus on few-shot imitation learning, which enables the systems to learn new tasks from just a handful of demonstrations [28], [29].

2) Language-Conditioned IL: Language-conditioned imitation learning is an important advancement in this field. It combines natural language instructions with demonstration-based learning to create more flexible and intuitive robot control. This approach allows users to guide robots through natural language commands [8], [30]–[32].

3) LLM-Generated Demonstrations: Robot demonstration generation is an important concept in the stream of research. LLMs have been used to automatically generate simulations for different purposes [33], [34]. Gensim [3] introduced an approach that leverages LLMs to automatically generate diverse robotic simulation environments and expert demonstrations. However, this approach has limitations for a wide range of tasks, such as long-horizon tasks or those requiring precise state awareness. Our approach expands the domain of solvable tasks by utilizing LLM-generated finite state machines as guides for demonstration collection.

III. Method

A. SMSL Generation via LLM-Based States, Operations and Transitions Filtering

The proposed framework uses State Machine Serialization Language (SMSL) to guide demonstration collection for robot manipulation tasks. SMSL is a data language that stores

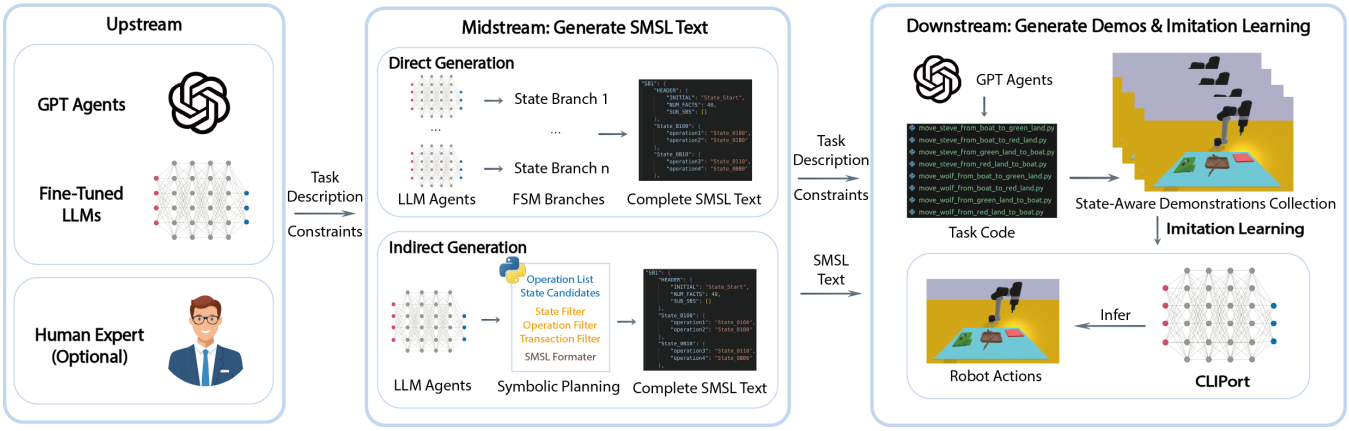


Fig. 3: Overall architecture of the proposed method. The upstream processes the high-level task goal to provide detailed task descriptions with constraints to the midstream. We propose two methods in the midstream: The direct generation is to split the whole SMSL generation into multiple state branches to be generated separately and then integrate them. Indirect generation will use LLM to generate a symbolic planning script to plan and format an SMSL text. The LLM agents in downstream with engineered prompts will take this SMSL text along with the task description and constraints from the upstream to generate the task demonstration code. We will then use the task code to collect the dataset for imitation learning in a "state-aware" workflow.

the definition of FSMs in a text format, which can represent the workflow of tasks [19]. While LLMs could potentially generate plans for complex tasks directly, creating comprehensive planning for complex tasks with a large number of states and transitions from unstructured descriptions is still challenging and lacks scalability. To address this limitation, we implement an indirect SMSL text generation approach that utilizes LLMs to generate and complete executable scripts. The generation process has the following steps:

First, the LLM analyzes the task description to list all independent operations (e.g., "move Object A to Position B") and find an optimized symbolic representation for the states based on the task requirements (e.g., "State_A"). Second, the LLM defines the state representation and state space, then generates a script to exhaustively enumerate all possible state configurations, without considering the task constraints. These states serve as candidates for state filtering. Third, based on the constraints of the task, the LLM constructs filter functions to rule out states that violate the constraints. After filtering, the LLM will list all valid state-operation pairs and add an operation filter script to prevent transitions from invalid states. It then writes a script to infer state transitions based on the valid state-operation pairs, which will lead to the goal state of this transition. The system applies the state filter again to verify if these goal states are valid; if not, then it removes the entire transition. Finally, the LLM formats the remaining transitions into SMSL text.

The overall algorithm for this LLM-based SMSL generation process is presented in Algorithm 1, with a mathematical formulation of the filter representation for a river-crossing puzzle provided in the Appendix.

B. State-Aware Task Demonstration Code Design

Existing approaches like GenSim generate demonstration code for the task that initializes the environments to random object placement [3], [4], which will introduce discrepancies between the task description and the demonstration when applied to longer horizon, more complex tasks. An example of this format of tasks is given in Listing 1 in Appendix. We design our state-aware task code to establish environment-state persistence via decoupled environment initialization from task logic. Listing 2 in Appendix illustrates what our modified task format looks like. It features three improvements:

Randomize Environment Under Constraints: In our workflow, a randomized environment will be created for the initial state under the constraints defined in the setup script, guaranteeing compliance with symbolic state requirements.

Environment Passing: Rather than resetting the environment and spawning the objects in the task code, we will directly pass the environment to the task code.

Deterministic Object Referencing: To ensure that objects across multiple episodes are referenced deterministically, we create a persistent dictionary which maps each entity (URDF handle) to a runtime object ID.

These changes ensure two functionalities can be performed correctly: First, they free LLMs from locking on low-level objects instantiation towards high-level task goal and constraints understanding. We also expose entity handles to LLMs and allow it to inspect geometric feature when necessary (e.g., chessboard blocks or arbitrary slots), and calculate placement offsets from a target object's pose. Second, it preserves dataset diversity via randomizing all initial pose configurations, while ensuring the state configurations are consistent to the object poses in the environment.

C. Prompt Engineering for Task Generation Agents

Our code generation pipeline transforms symbolic operations into executable task demonstration scripts by leveraging

Algorithm 1 Indirect SMSL Generation Using LLM

Require: Task description T, state constraints $\mathcal{C}_{state}$, operation constraints $\mathcal{C}_{op}$

Ensure: Valid states, operations, transitions mapping

Phase 1: Generate Operations and State Candidates

$O \leftarrow$ LLM generates operations from T

$S \leftarrow$ LLM generates script to list all states using Cartesian product from T as candidates, ignoring constraints

Phase 2: Filter States

$S_{valid} \leftarrow \{s \in S \mid$

LLM adds filter that validates s against $\mathcal{C}_{state}\}$

Phase 3: Assign Operations

for $s \in S_{valid}$ **do**

$O_s \leftarrow$ LLM adds filter that validates operations from O for s

end for

Phase 4: Filter Transitions

for $(s, o) \in S_{valid} \times O_s$ **do**

$s' \leftarrow$ LLM adds code to apply operation o to state s

if s' violates $\mathcal{C}_{state}$ or o violates $\mathcal{C}_{op}$ **then**

Remove transition (s, o)

end if

end for

Phase 5: Generate Output

Convert valid transitions to JSON format

return SMSL in JSON format

multiple LLM agents to use our structured prompts. The first stage of the process is broken into two subcomponents, and the second and third stages are recursively applied to each of the operations in the SMSL text:

Agent 0 / Human: Initial-State Environment Setup Script

An LLM agent will process task descriptions, spatial constraints from expert knowledge and scene entity library references to produce the initialization scripts that:

- Spawn objects at random poses under geometric constraints *only* for the initial state.
- Establish persistent mappings between scene entities and simulation object IDs.

Verifications can be made to ensure this initialization script can have the spawned environment consistent with the initial state's configurations before passing it to the next LLM agent.

Agent 1: Task Specification Generation

Another LLM agent will read through this initial state environment setup script and task description to generate a structured task specification in JSON format containing:

- Entity List: Categorized URDF file names of used meshes by their physics types (fixed or rigid).
- Entity to Inspect: A URDF file name that requires geometric inspection by the task generation agent.
- Natural language task summary.

Agent 2: Task Code Generation

The final LLM agent will read through CLIPort's API to understand the definition of the "task" class, then to review

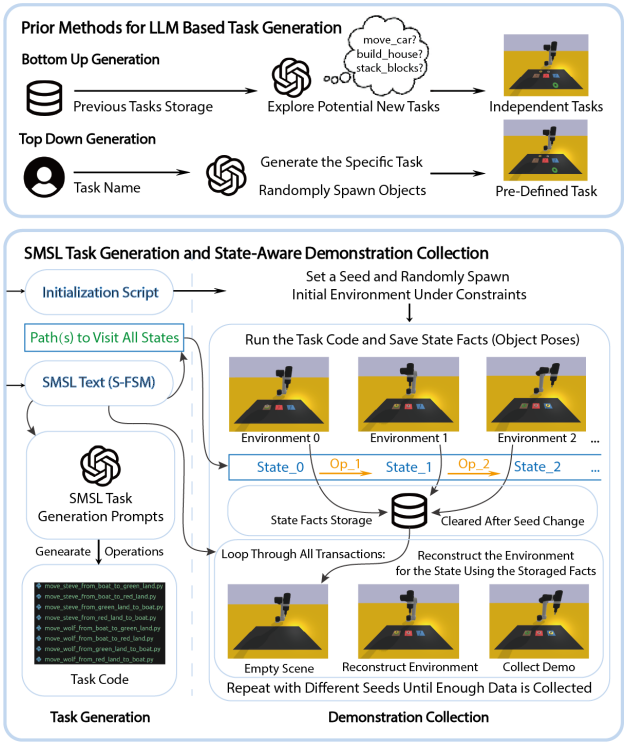


Fig. 4: Comparison of demonstration generation workflows. The existing bottom-up approach is to let LLM to explore potential new tasks from the previous task list. The existing top-down approach will use the user-defined task name and generate code for this specific task. Both of these methods will reset the environment and spawn objects randomly on the tabletop when generating a demo, which are not suitable for long-horizon tasks or tasks with dynamically evolving spatial constraints.

examples of success and failure with the type of failure. It creates the task code according to the JSON description extracted from the task specification and also inspects the URDF if it's listed in "entity to inspect", so that it knows how to properly set the offset for the target pose.

D. Demonstration Dataset Generation

Our framework provides automatic demonstration dataset generation in a two-phase approach using the constructed FSM, the workflow and the comparison with the existing LLM-centralized workflow for task generation is illustrated in Fig. 4. At first, we apply graph search approach in order to find either a single path that covers all the states or, if that fails, generate multiple minimum-length paths to cover all states: We employ a depth-first search that maximizes state coverage; the algorithm seeks to find a single path containing every state by tracking visited states with bitmasking and if failed, produces several minimum-length paths using a coverage-optimized queue. These paths provide guidance for exploration of states, which also infers the sequence of operations applied to the initial environment.

The second stage uses the paths to generate training demonstrations for CLIPort [8]. It starts with spawning an

environment of the initial state and storing the accurate state configurations in the storage. Then, it applies the first operation in the path to the environment and collects the state configurations as well. This process is then repeated until it has collected all or exemplary state configurations in case of exceedingly large state space. Following this, the framework loops over all of the transitions as defined in the SMSL, reconstructs the environment from the stored configurations, performs the operation, and generates the demonstration. To be more specific, we denote the object poses for each transition $(s_i, o_j \rightarrow s_k)$ in the path as:

$$F_{s_i} = \{e \mapsto \text{get\_object\_pose}(id_e) \mid e \in E\} \quad (1)$$

where E represents the set of all entities (objects) in the environment, "get_object_pose" is a script function that will take the object id as input and output the object pose. During each transition $(s_i, o_j \rightarrow s_k)$, the environment is initialized with constraint-compliant random poses, followed by saving the object poses as state configurations F_{s_i}. The system then executes operation o_j using LLM-generated task scripts with the current environment and save the resulting state configurations as F_{s_k}. This process continues along the paths until all state configurations are collected, where these configurations F_s are stored in a state dictionary unique to each randomized initialization. This approach enables deterministic state reconstruction and effectively separates the demonstration collection process from environment randomization.

The demonstration collection iterates through all transitions in SMSL (in any order, allowing parallel execution), where for each transition, the system spawns an environment using stored state configurations, executes the corresponding task code, and collects the required data for CLIPort (RGB-D observations, language goals, and actions). The environment is then initialized in random poses that are consistent with constraints.

E. Design of Experiments

The experimental evaluation includes three complex tabletop manipulation tasks, each representing a distinct finite state machine (FSM) with varying complexity levels according to Fig. 1.

Towers of Hanoi: The set-up includes three stands (blue, red, and brown) and three rings: gray, yellow, and green in increasing radius. Following standard rules, only smaller rings can be placed on larger ones, the robot can only move one of the rings at a time. The three stands are spawned randomly on the table while the rings are all located on the brown stand.

River Crossing: A puzzle with the Human, Wolf, Sheep, and Grass entities that initially begin with all entities on red land. Sheep cannot be alone with either Wolf or Grass in the absence of Human. For any two-object boat transport, Human must be present. A detailed task description and constraints representation are provided in the Appendix.

Chess: The Chess Board puzzle has nine blocks and two chess pieces. Among the nine blocks, three of them are impassable. Pieces can be placed on top of each other in the same block, and the state "star chess is on top of the circle chess" is different from the state "circle chess is on top of the star chess". At the start of the game, the star chess is spawned at the bottom left block as seen in Fig. 1, and the circle chess is spawned at the bottom right block. The goal is to switch the positions of the pieces.

IV. Results and Discussion

While the solution path is solved by a shortest path finding algorithm, our approach emphasizes both the final outcomes and the intermediate steps, recognizing that each transition provides valuable insight into the overall task structure. Unlike puzzle-solving tasks that aim solely to reach a predefined final configuration (the "goal state"), our implementation emphasizes the importance of all meaningful states along the way. Intermediate configurations, those that occur between the start and goal, are treated as equally valuable. This reflects real-world scenarios, where achieving any valid or informative state can be useful, and tasks often involve navigating through many such states. As a result, our data collection strategy intentionally captures a wide spectrum of these intermediate states, not just the end solutions.

We evaluated our method by training on our collected dataset across three puzzles. We trained the model with 200, 500, and 1000 demonstrations for each operation, and we used the average success rate across all operations as the evaluation metric. To highlight the necessity of state-aware demonstration collection for these complex long-horizon tasks, we compared our method against a control condition where all objects are randomly placed on the table when collecting demos - a typical setting in previous methods. We also trained with 200, 500, and 1000 demonstrations per operation for this control condition and tested in the same puzzle-solving contexts.

For multi-agent task specification, we used GPT-4o to generate complete task code. The average generation times were 13.60 seconds for the Hanoi task, 17.96 seconds for the River-Crossing task, and 15.48 seconds for the Chess task. All experiments were carried out on a workstation running Ubuntu 20.04, equipped with an Intel Core i9-13900K CPU, 64 GB of DDR5 RAM, and a GeForce RTX 4090 GPU. We trained each model for 10 epochs and evaluated the best checkpoint. Test results demonstrate that the policy trained on random object placement demos does not generalize well to situations where objects need to understand dynamically changing state dependencies. Table II presents the detailed comparative results.

We show that with LLM-based simulation generation, our proposed method can successfully perform long-horizon complex tasks while the control condition (randomly placing objects on the table) fails. Our contributions allow LLM-powered robots to be capable of performing real world tasks where environmental states must be rigorously tracked and propagated across long-horizon complex tasks.

TABLE II: Performance Comparison Across Tasks and Demo Numbers (%).

Method	Number of Demonstrations		
	200	500	1000
Hanoi (tolerance 0.01m, 15 degrees)			
Our Method	84.9	94.2	98.0
GenSim	58.7	60.6	58.0
River Crossing (tolerance 0.06m, 15 degrees)			
Our Method	71.6	94.7	96.2
GenSim	2.7	7.7	5.5
Chess (tolerance 0.01m, 15 degrees)			
Our Method	89.0	92.9	98.0
GenSim	39.4	44.3	43.0

V. Conclusion

Current demonstration generation approaches using LLMs in imitation learning frameworks are limited by the coverage of the sample set. When the demonstration set lacks examples for dynamically evolving situations, the model may fail to generalize, leading to cascading errors during robotic execution. Our experiments confirm that existing frameworks are particularly prone to such failures in long-horizon tasks. To address this limitation, we propose a state-aware approach that enhances demonstration coverage. Our implementation improves the success rate of long-horizon tasks, achieves an up to 98% success rate in these tasks, compared to the controlled condition for the existing approach, which only had a success rate of up to 60%, and some tasks experience catastrophic failure. The research of CLIPort [8] and GENSIM [3] already provides us a solid foundation of sim-to-real capabilities for our work; accordingly, we focus here on advancing long-horizon task sequencing and automated code generation rather than revalidating transfer performance. In future studies, we plan to conduct real-world imitation learning experiments and evaluate the SMSL framework on physical hardware, incorporating a broader range of learning architectures.

Appendix

A. Formalism of the River Crossing Puzzle

To demonstrate our methodology, here we use the river crossing task as an example, which is a task with constraints that requires reasoning while robot moving. The game involves four entities—Human, Sheep, Wolf, and Grass. They must be transported from red land to green land via a boat with limited capacity. We define the following constraints to the task. 1) The robot can move one entity per operation between adjacent locations (red land - boat - green land). 2) Sheep cannot coexist with Wolf or Grass without the Human presence. 3) The boat holds no more than two entities, requiring Human's presence when occupied by two. We now present our LLM-based symbolic planning approach, through the following formalization steps.

1) State Space Definition: We formalize the problem as a 4-tuple $T = (P, O, A, C)$ where:

- 1. $P = \{\text{Red\_land}, \text{Boat}, \text{Green\_land}\}$ (Positions)
- 2. $E = \{\text{Human}, \text{Sheep}, \text{Wolf}, \text{Grass}\}$ (Entities)
- 3. O: 16 possible move operations between positions
- 4. C: Safety and capacity constraints

The complete state space is generated through a Cartesian product:

$$\mathcal{S}_{\text{candidate}} = \prod_{l \in L} S = 3^4 = 81 \text{ states} \tag{2}$$

Each state is encoded as:

$$s = (s_{\texttt{Human}}, s_{\texttt{sheep}}, s_{\texttt{wolf}}, s_{\texttt{grass}}) \in S^4 \tag{3}$$

2) States Filtering: Here for this river-crossing task, we define the constraint as $C = C_{\text{safety}} \wedge C_{\text{capacity}}$:

$$\begin{aligned} &\mathcal{C}_{\text{safety}} : \forall e \in E \setminus \{\text{human}\}, \\ &(s_e = s_{\text{sheep}} \wedge e \in \{\text{wolf}, \text{grass}\}) \quad \rightarrow s_{\text{human}} = s_{\text{sheep}} \end{aligned} \tag{4}$$

$$\begin{aligned} \mathcal{C}_{\text{capacity}} : &\sum_{e \in E} \mathbb{I}(s_e = \text{boat}) \leq 2 \\ &\wedge \Big(\sum_{e \in E} \mathbb{I}(s_e = \text{boat}) = 2 \qquad \rightarrow s_{\text{human}} = \text{boat}\Big) \end{aligned} \tag{5}$$

where $\mathbb{I}$ is the indicator function that equals 1 when the condition is true and 0 otherwise, $\mathcal{C}_{\text{safety}}$ corresponds to the second constrains and $\mathcal{C}_{\text{capacity}}$ corresponds to the third constraint. For example, state $(\text{boat}, \text{green}, \text{boat}, \text{red})$ violates $\mathcal{C}_{\text{safety}}$ (sheep and wolf unsupervised), while $(\text{red}, \text{boat}, \text{boat}, \text{boat})$ violates $\mathcal{C}_{\text{capacity}}$ (3 entities in boat). The LLM agent will generate filter code that implements the above logic. After applying these constraints, the state space is reduced from 81 state candidates to 40 valid states.

3) Operations and Transitions Filtering: We here define a transition to be the complete path of a source state, an operation, and a resultant state. Valid transitions require operation-specific preconditions and post-validation. For a transition $t = (e, s_{\text{src}}, s_{\text{dest}})$:

$$\begin{aligned} \text{Pre}(o, s) = &(s_e = s_{\text{src}}) \wedge \\ &((s_{\text{src}} = \text{boat} \wedge s_{\text{dest}} \neq \text{boat}) \\ &\vee (s_{\text{src}} \neq \text{boat} \wedge s_{\text{dest}} = \text{boat} \\ &\wedge \mathcal{C}_{\text{capacity}}(s'))) \end{aligned} \tag{6}$$

where $\text{Pre}(o, s)$ is the precondition for operation o in state s, s_e is the location of entity e, s_{src} is the source location, s_{dest} is the destination location, and $\mathcal{C}_{\text{capacity}}(s')$ is a constraint ensuring the boat capacity isn't exceeded in the resulting state s'. This precondition states that an entity must be at the source location, and either we are moving from the boat to a land, or from a land to the boat (if capacity allows). For example, moving sheep from boat to green land requires: 1) Sheep is in boat, 2) Human remains with Wolf/Grass, and 3) Boat capacity remains valid. LLM agent generate executable filter script for the same logic, and after applying these LLM-generated filters, we can construct an accurate SMSL text for the FSM with much less human validation and correction compared to direct generation using LLM.

B. Task Demonstration Generation Code

```
# env is initialized with no objects
super().reset(env)
...
obj_pose = self.get_random_pose(env, size0)
obj_id = env.add_object(urdf, obj_pose, ...)
targ_pose = self.get_random_pose(env, size1)
targ_id = env.add_object(urdf, targ_pose, ...)
...
self.add_goal(objs=[obj_id], targ_poses=[
    targ_pose], ...)
```

Listing 1: GenSim Task Demonstration Code Format

```
# env is passed into the function
obj_id = env.asset_ids_dict['rigid']["x.urdf"]
targ_id = env.asset_ids_dict['fixed']["y.urdf"]
targ_pose = env.get_object_pose(targ_id)
self.add_goal(objs=[obj_id], targ_poses=[
    targ_pose], ...)
```

Listing 2: Our Task Demonstration Code Format

REFERENCES

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[3] L. Wang, Y. Ling, Z. Yuan, M. Shridhar, C. Bao, Y. Qin, B. Wang, H. Xu, and X. Wang, "Gensim: Generating robotic simulation tasks via large language models," *arXiv preprint arXiv:2310.01361*, 2023.

[4] P. Hua, M. Liu, A. Macaluso, Y. Lin, W. Zhang, H. Xu, and L. Wang, "Gensim2: Scaling robot data generation with multi-modal and reasoning llms," *arXiv preprint arXiv:2410.03645*, 2024.

[5] J. Wu, R. Antonova, A. Kan, M. Lepert, A. Zeng, S. Song, J. Bohg, S. Rusinkiewicz, and T. Funkhouser, "Tidybot: Personalized robot assistance with large language models," *Autonomous Robots*, vol. 47, no. 8, pp. 1087–1102, 2023.

[6] S. H. Høeg and L. Tingelstad, "More than eleven thousand words: Towards using language models for robotic sorting of unseen objects into arbitrary categories," in *Workshop on language and robotics at CoRL 2022*, 2022.

[7] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, "Code as policies: Language model programs for embodied control," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2023, pp. 9493–9500.

[8] M. Shridhar, L. Manuelli, and D. Fox, "Cliport: What and where pathways for robotic manipulation," in *Conference on robot learning*. PMLR, 2022, pp. 894–906.

[9] Y. Liu, Y.-C. Ku, J. Zhang, H. Ding, P. Kazanzides, and M. Armand, "dart vinci: Egocentric data collection for surgical robot learning at scale," *arXiv preprint arXiv:2503.05646*, 2025.

[10] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[11] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning," in *Proceedings of the international conference on automated planning and scheduling*, vol. 30, 2020, pp. 440–448.

[12] Y.-q. Jiang, S.-q. Zhang, P. Khandelwal, and P. Stone, "Task planning in robotics: an empirical comparison of pddl-and asp-based systems," *Frontiers of Information Technology & Electronic Engineering*, vol. 20, pp. 363–373, 2019.

[13] J. A. Baier, F. Bacchus, and S. A. McIlraith, "A heuristic search approach to planning with temporally extended preferences," *Artificial Intelligence*, vol. 173, no. 5-6, pp. 593–618, 2009.

[14] J. Hoffmann, "Ff: The fast-forward planning system," *AI magazine*, vol. 22, no. 3, pp. 57–57, 2001.

[15] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.

[16] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman *et al.*, "Do as i can, not as i say: Grounding language in robotic affordances," *arXiv preprint arXiv:2204.01691*, 2022.

[17] P. A. Jansen, "Visually-grounded planning without vision: Language models infer detailed plans from high-level instructions," *arXiv preprint arXiv:2009.14259*, 2020.

[18] S. S. Kannan, V. L. Venkatesh, and B.-C. Min, "Smart-llm: Smart multi-agent robot task planning using large language models," in *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2024, pp. 12 140–12 147.

[19] Y. Liu and M. Armand, "A roadmap towards automated and regulated robotic systems," *arXiv preprint arXiv:2403.14049*, 2024.

[20] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 1470–1477.

[21] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar *et al.*, "Inner monologue: Embodied reasoning through planning with language models," *arXiv preprint arXiv:2207.05608*, 2022.

[22] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg, "Progprompt: Generating situated robot task plans using large language models," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2023, pp. 11 523–11 530.

[23] T. Osa, J. Pajarinen, G. Neumann, J. A. Bagnell, P. Abbeel, J. Peters *et al.*, "An algorithmic perspective on imitation learning," *Foundations and Trends® in Robotics*, vol. 7, no. 1-2, pp. 1–179, 2018.

[24] H. Ravichandar, A. S. Polydoros, S. Chernova, and A. Billard, "Recent advances in robot learning from demonstration," *Annual review of control, robotics, and autonomous systems*, vol. 3, no. 1, pp. 297–330, 2020.

[25] S. Schaal, "Is imitation learning the route to humanoid robots?" *Trends in cognitive sciences*, vol. 3, no. 6, pp. 233–242, 1999.

[26] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and autonomous systems*, vol. 57, no. 5, pp. 469–483, 2009.

[27] A. Billard, Y. Epars, S. Calinon, S. Schaal, and G. Cheng, "Discovering optimal imitation strategies," *Robotics and autonomous systems*, vol. 47, no. 2-3, pp. 69–77, 2004.

[28] Y. Duan, M. Andrychowicz, B. Stadie, O. Jonathan Ho, J. Schneider, I. Sutskever, P. Abbeel, and W. Zaremba, "One-shot imitation learning," *Advances in neural information processing systems*, vol. 30, 2017.

[29] C. Finn, T. Yu, T. Zhang, P. Abbeel, and S. Levine, "One-shot visual imitation learning via meta-learning," in *Conference on robot learning*. PMLR, 2017, pp. 357–368.

[30] E. Jang, A. Irpan, M. Khansari, D. Kappler, F. Ebert, C. Lynch, S. Levine, and C. Finn, "Bc-z: Zero-shot task generalization with robotic imitation learning," in *Conference on Robot Learning*. PMLR, 2022, pp. 991–1002.

[31] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, X. Chen, K. Choromanski, T. Ding, D. Driess, A. Dubey, C. Finn *et al.*, "Rt-2: Vision-language-action models transfer web knowledge to robotic control," *arXiv preprint arXiv:2307.15818*, 2023.

[32] O. M. Team, D. Ghosh, H. Walke, K. Pertsch, K. Black, O. Mees, S. Dasari, J. Hejna, T. Kreiman, C. Xu *et al.*, "Octo: An open-source generalist robot policy," *arXiv preprint arXiv:2405.12213*, 2024.

[33] A. Zeng, M. Liu, R. Lu, B. Wang, X. Liu, Y. Dong, and J. Tang, "Agenttuning: Enabling generalized agent abilities for llms," *arXiv preprint arXiv:2310.12823*, 2023.

[34] M. Hu, P. Zhao, C. Xu, Q. Sun, J. Lou, Q. Lin, P. Luo, S. Rajmohan, and D. Zhang, "Agentgen: Enhancing planning abilities for large language model based agent via environment and task generation," *arXiv preprint arXiv:2408.00764*, 2024.