Robot Operating Home Appliances by Reading User Manuals

Jian Zhang, Hanbo Zhang, Anxing Xiao, David Hsu
School of Computing & Smart System Institute
National University of Singapore
corresponding to zhang.jian@u.nus.edu

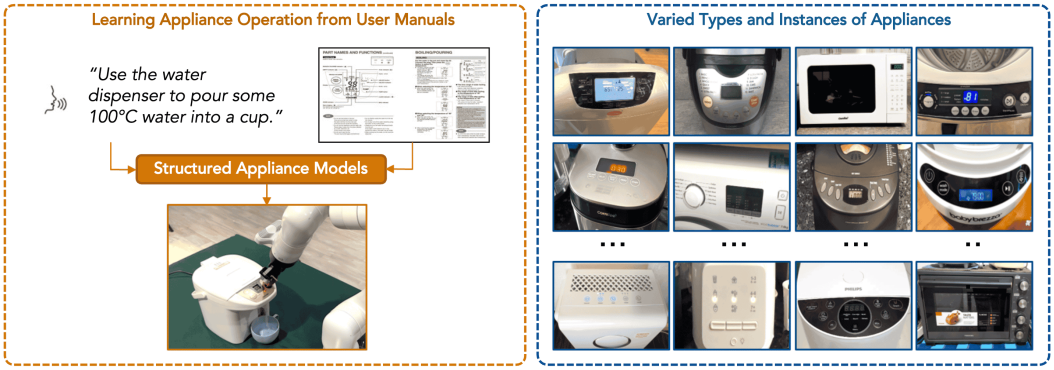


Figure 1: ApBot enables robots to operate **diverse**, **novel**, **complex** home appliances in a zero-shot manner using manuals. It translates open-ended instructions to grounded multi-step actions.

Abstract: Operating home appliances, among the most common tools in every household, is a critical capability for assistive home robots. This paper presents ApBot, a structured appliance modelling approach that enable robots operate *novel* household appliances by "reading" their user manuals. ApBot faces multiple challenges: (i) infer goal-conditioned partial policies from unstructured, textual descriptions in the manual document, (ii) ground the policies to the appliance in the physical world, and (iii) execute policies reliably over potentially many steps, despite compounding errors. To address these, ApBot constructs a structured, symbolic model of an appliance from its manual, with the help of a large vision-language model (VLM). It grounds symbolic actions to control panel elements and updates the model based on visual feedback. Our experiments show that across a wide range of simulated and real-world appliances, ApBot achieves consistent, statistically significant improvements in task success rate, compared with state-of-the-art large VLMs used directly as control policies. These results suggest that structured representations of appliance models are crucial for robust robot operation of home appliances, especially complex ones. Code is provided.

Keywords: Home Appliance Operation; Structured Model for Decision Making; Foundation Models for Robotics

1 Introduction

Operating household appliances is a fundamental yet underexplored topic for assistive robots at home. It could greatly expand robot capabilities. Unlike passive tools, appliances encapsulate complex, high-level functionalities (e.g., cooking, cleaning, heating) that direct manipulation cannot replicate. Ideally, robots should operate general-purpose appliances automatically using maunals.

9th Conference on Robot Learning (CoRL 2025), Seoul, Korea.

However, this task challenges existing robotic systems. First, manuals are unstructured text-symbol documents that are hard for robots to interpret. In addition, appliances follow constrained, mode-based workflows that tolerate little error. Therefore, this work seeks to answer: *How can we enable robots to generate visually grounded policies for novel appliance operation with user manuals?*

To this end, we propose ApBot, a structured appliance modelling framework for operating generalizable, open-world appliances (Fig 1). Using large vision-language models (LVLMs), ApBot constructs symbolically structured models of novel appliances from user manuals for interpretable, controllable, and verifiable policy generation. The model captures symbolic representations of executable actions, appliance states, and transition rules. Given a natural language task, the model generates an action sequence grounded in the panel layout, which is then executed by low-level skill primitives. To address manual ambiguity and open-world issues [1], ApBot iteratively updates the models based on visually observation to better reflect real appliance behavior. As a result, ApBot robustly operate novel, complex appliances with language instructions and no additional training.

Structured representation with online error correction significantly improves robustness, especially for complex appliances. To evaluate ApBot, we constructed a simulated benchmark from manuals [2, 3], covering 6 appliance types and 30 interactive instances; *dehumidifier*, *bottle washer*, *rice cooker*, *microwave oven*, *bread maker*, and *washing machine*. Each appliance is paired with 10 natural language instructions. Since appliances vary in complexity, we standardize the number of variables (e.g., *time* and *temperature*) required by the instructions for each type, ranging from easy to hard. We compare ApBot with LLM- and VLM-based methods [4–6], and the results show our method consistently outperforms them by a clear margin. We also deploy ApBot in the real world with a Kinova Gen3 arm and validate its effectiveness on real appliances.

In summary, we propose a novel, symbolically structured representation for generalizable home appliance operation. It bridges the gap between unstructured inputs and policies, enabling robots to make controllable decisions and reliably operate novel appliances using visual input and manuals.

2 Related Work

Robotic Appliance Operation Previous work related to robotic appliance operation falls into three categories: perception, low-level manipulation, and long-horizon plan generation. For perception, existing works mainly concentrate on button localization, by leveraging edge-based visual features [7, 8], RFID tags [9], or neural networks [10–13]. For low-level skills, human-machine interfaces typically rely on a limited and shared set of input modalities [14] (e.g. keyboards, dials, swipes, pinches). Button manipulation is important, hence, many works focus on physical interaction with diverse button types by specializing fingertips [15] or leveraging additional sensory feedback [15, 16]. Alternatively, language-conditioned policies use linguistic embeddings to speed up adaptation [17]. Long-horizon appliance operation remains relatively underexplored. A recent work [12] uses handcrafted behaviors to generate plans for home appliance operation, but lacks generalization to novel appliances or tasks. By contrast, our approach learns to model the appliances by reading the user manuals, hence enabling operation policy generation for new appliances and task instructions in a zero-shot manner.

Foundation Models for Robotic Decision Making Foundation models have been widely applied to robotic decision-making [18]. Prompting large models to generate structured representations, such as logic frameworks [19–22] and codes [23–28] enables integration with external solvers or executors, though often under strict syntax constraints. It has been shown to improve reasoning precision at the cost of generality, particularly for long-horizon tasks with hard constraints [29–32]. Unstructured representations such as textual reasoning trees [33–36] or natural languages [37–42] offer flexibility and generality but suffer from ambiguity and lack correctness guarantees. Approaches such as syntax validation [43, 44], corrective feedback [40, 45], and prompt optimization [46] aim to take advantage of both by modular design.

Particularly, domain-specific languages (DSLs) can enhance LVLM reliability and robustness via in-context learning [25, 47–50], offering key insights for our structured design for appliances. We

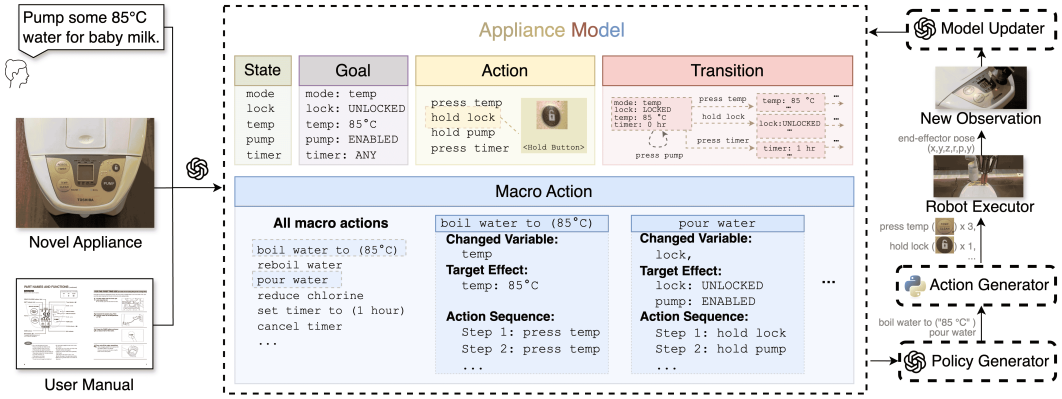


Figure 2: Overview of ApBot. The structured model built from manuals can generate actions to operate novel appliances. It can be calibrated with observed feedback during close-loop execution.

adopt a DSL-based design tailored to appliance operation and generate it via in-context learning and syntax validation. To support numerical computation, we enable the invocation of external function calls similar to [24]. This ensures generalizability while maintaining correctness guarantees.

Graphical User Interface Agents Graphical User Interface (GUI) agents [51, 52], similar but unlike appliance operation, operate devices through software interfaces. Small-scale multimodal models can be trained from scratch to ground manuals with observations and goals for policy learning [53, 54]. Recently, the prevailing approach involves fine-tuning VLMs on datasets of web or mobile interactions, such as clicking, typing, or tapping, to directly predict on-screen actions [55–58]. An alternative line of work uses visual prompting, where VLMs select actions based on overlaid marks (e.g., boxes or numbers) on GUI screenshots [59–62]. Yet, these methods often struggle with complex GUIs, especially when the documentation is incomplete or unstructured. Another line of related research is Retrieval-Augmented Generation (RAG) [63], which augments model input with information retrieval from external sources, such as online manuals or documents. Building on this idea, we introduce structurally grounded representations for appliance operation, which encode knowledge from user manuals to enable robust plan generation.

3 ApBot

3.1 Problem Formulation

We consider the task of household appliance operation based on parameterized natural language instructions and visual observations, with the help of textual information from user manuals. Given a natural language instruction (e.g., *Cook long grain rice for 1 hour*), the system must interpret the goal state, reason about appliance constraints, and generate a sequence of executable low-level actions that complete the task.

We formulate the appliance as a state machine [64] with tasks specified as a subset of goal states: $\mathcal{M} = \langle S, A, \mathcal{T}, S_g \rangle$. Each state s in S comprises a list of variables for the appliance (e.g., a rice cooker with variables such as power (on/off), menu (rice/porridge/soup), timer (1–6 hours)). For action space A, we empirically find that most actions of operating a home appliance can be categorized into two classes: A_g and A_n. Any action in A_g will directly *go* to a pre-defined, specific state (e.g., press "Menu" to select cook menu), if the current state is in a specific subset of S (e.g., the rice cooker is powered on and not locked). Any action in A_n (e.g., "+" and "-") will turn the state of a variable to a neighborhood value, subject to the current state. Accordingly, we have a deterministic transition model for each type $s' = \mathcal{T}(s, a)$, where $\mathcal{T} \in \{\mathcal{T}_g, \mathcal{T}_n\}$. Given the above formulation, we aim to find a shortest sequence of actions $\mathbf{a}^* = (a_1^*, a_2^*, ..., a_T^*)$ to achieve an arbitrary goal state in S_g, a subset of S.

However, in practice, we cannot have access to the underlying true model $\mathcal{M}$. Instead, we construct an approximated one $\overline{\mathcal{M}} = \langle \overline{S}, \overline{A}, \overline{\mathcal{T}}, \overline{S}_g \rangle$ from the manual, upon which we generate the operation

policy for the appliance. To construct $\overline{\mathcal{M}}$, we assume that robots can observe the home appliances visually and have the corresponding manuals in raw text. We assume that the users will interact with robots in natural languages to specify tasks, which defines $\overline{S}_g$ given $\overline{S}$. We posit that LVLMs can read raw textual information in the manual and build a partially correct appliance model $\overline{\mathcal{M}}$ accordingly, by proper design.

We propose a system, ApBot, for natural language control of household appliances by combining user manuals and visual input. As shown in Figure 2, the system (1) constructs a symbolic model from manuals (Sec. 3.2), (2) grounds symbolic actions to visual control elements (Sec. 3.3), (3) models transitions as macro actions (Sec. 3.4), and (4) executes tasks with closed-loop updates based on real-time feedback (Sec. 3.5) to address errors of model construction.

3.2 Construct Structured Appliance Model

Modeling States, Actions, Transitions. To construct the symbolic model $\overline{\mathcal{M}} = \langle \overline{S}, \overline{A}, \overline{\mathcal{T}}, \overline{S}_g \rangle$, we sequentially generate the state space $\overline{S}$, action space $\overline{A}$, and transition function $\overline{\mathcal{T}}$ with the help of LVLM agents [6] using prompting [4]. Concretely, for each of them, we provide the manual, a predefined output format, and a complete list of valid options, and ask LVLMs to generate the appliance model accordingly. Model generation goes in an in-context way [65], where examples are provided in the input to improve robustness. Syntax checkers are then applied to ensure output validity, with up to three regeneration attempts if errors occur, e.g, violations of constraints. We provide a detailed example of appliance modeling in Appendix A, along with the corresponding prompts in Appendix I, and list the syntax checks used for validation in Appendix B.

Extracting Goals from Instructions. To infer the goal state $\overline{S}_g$ from a natural language instruction, we prompt the LVLM agent to produce a partial assignment over symbolic variables that fulfills the task requirements. For example, given the instruction "Cook long grain rice for 1 hour", the inferred goal corresponds to a symbolic state where power is on, menu is long grain rice, and timer is 1 hour, while all other variables remain unconstrained.

3.3 Action Grounding

To make actions physically executable, we need to ground the symbolic actions visually onto the observed *control panel elements*, which are interactive components of an appliance, such as buttons, dials, and printed touch pads. Each grounded action $\hat{a}$ is a tuple $\hat{a} = (a, b, \sigma)$, where $a \in \overline{A}$ is the symbolic action from the manual, b is a bounding box of the visual region, and $\sigma \in \{\texttt{press}, \texttt{hold}, \texttt{turn}\}$ denotes the primitive robot skill required to execute the action. We demonstrate the pipeline of action grounding in Fig. 3.

Control Element Detection. We assume the control panel elements can be clearly detected using existing object detectors. We prioritize high recall in detection to ensure all buttons are captured. To this end, we run three models in parallel. Segment Anything (SAM) [66] segments the image into regions of visually distinct entities. OWL-ViT2 [67] is queried with prompts of "button", "dial", and "switch" to detect control elements. An OCR model [68] extracts regions of visible text labels. We take the union to form a candidate set for all control elements. To remove false positives, we sort the bounding boxes in descending order of detection confidence. For each pair of boxes (b_i, b_j), if $\text{IoU}(b_i, b_j) > 0.85$, we discard the box with lower detection confidence, as overlapping boxes likely refer to the same object. We further use LVLMs to check whether each remaining box likely contains a valid control panel element, following [69]. So far, we have a set of boxes $\mathbf{B} = \{b_i\}_{i=1}^{N_b}$, where N_b is the number of boxes.

Actions Grounding. To get the executable action $\hat{a} = (a, b, \sigma)$, we need to do visual grounding, i.e., an injection from symbolic actions $\overline{A}$ to boxes $\mathbf{B}$, and identify the manipulation type σ for each $a \in \overline{A}$. To do so, we first query LVLMs to assign an action $a \in \overline{A}$ for each $b \in \mathbf{B}$, i.e., a mapping from $\mathbf{B}$ to $\overline{A}$. Inversely, now, each a may: (1) have a unique box b; (2) have no box; (3) have a set of boxes $\mathbf{B}_a \subseteq \mathbf{B}$, $|\mathbf{B}_a| > 1$. For (1), it is ideal. For (2), we directly remove it from $\overline{A}$ since it is no longer executable, hence, all tasks involving this action will fail. For (3), we impose the following two heuristics by further prompting the LVLMs: 1. the box including clear physical boundaries is

preferred; 2. the box with an icon-based label is preferred over text-only ones. Finally, we assign the manipulation type σ directly based on the description of a along with the assigned visual region b, forming $\hat{a} = (a, b, \sigma)$, which can be executed using the corresponding low-level primitive skill.

3.4 Structured Transition Modeling with Macro Actions

Perfectly modeling appliances with one shot is impractical due to the inherent ambiguity of the manuals (e.g., "adjust to desired level") and inevitable errors of LVLMs (e.g., hallucinations), making planning hard. Instead, we leverage two observations of common appliance designs: (1) variable adjustments often follow consistent action sequences, such as pressing "+" repeatedly or entering digits conditioned on an explicitly specified number string; and (2) user manuals often describe step-by-step tutorials for common usages. We formalize user manuals as macro actions $\overline{\Phi}$, a constructed version of the underlying ground truth set of macro actions Φ, for efficient and generalizable appliance modeling.

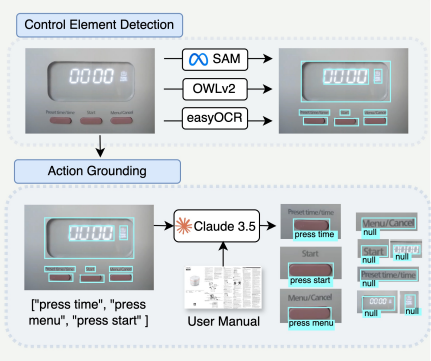


Figure 3: Overview of action grounding with visual observations.

Definition of Macro Actions. Macro action $\phi \in \Phi$ is a parameterized sequence of symbolic actions that encapsulate a meaningful functionality (e.g., `Cook`, `Kitchen Timer`). Specifically, each macro action ϕ consists of a symbolic action sequence $\mathbf{a}_\phi = (a_1, a_2, ...)$, where $a_i \in A$. Besides, each macro action has a set of variables s_v on which it imposes effects and a macro transition describing these effects $\Gamma(s_v)$. Note that s_v is part of the full state s. Formally, $\phi = (s_v, \Gamma(s_v), \mathbf{a}_\phi)$. For example, the macro action $\phi =$ `Cook(LongGrain, 1 hour)` may include two symbolic actions: $a_1 =$ `press_menu` and $a_2 =$ `press_time`, which will change the values of variables $s_v =$ (`menu`, `cooking time`), so as to (1) set the menu to `LongGrain`; and (2) set the cooking time to `1 hour`. Macro actions enable LVLMs to plan by specifying high-level subgoals, simplifying reasoning by substantially reducing the reasoning horizon.

Modeling Macro Actions. The full list of macro actions, including the corresponding variables and target effects, is extracted by prompting LVLMs (see Appendix I). To fully model the macro actions, we need to generate a sequence of low-level actions $\mathbf{a}_\phi$ based on the current variable s_v and target transition $\Gamma(s_v)$. Specifically, $\mathbf{a}_\phi$ can be directly computed from two transition types: $\overline{\mathcal{T}}_n$ for A_n actions like "+" or "-", and $\overline{\mathcal{T}}_g$ for go-to actions in $\overline{A}_g$. The computation invokes codes directly, similar to [24], based on the current and goal values. For actions in A_n, we compute the number of repeats based on its transition model (e.g., from 1 to 5 requires 4 presses of "+"). For actions in A_g, we translate the description in the manual directly to get the specific actions.

3.5 Closed-loop Task Execution

Automatic Execution from Macro Actions. To specify the executable actions for robots, ApBot generates a parameterized symbolic task policy $\pi = [\phi_1, \phi_2, ..., \phi_T]$ conditioned on the inferred goal $\overline{S}_g$, where each $\phi_i \in \overline{\Phi}$ is a parameterized macro action, covering one or more variables in goal state. To do so, we feed the list of applicable macro actions and the textual specification of $\overline{S}_g$ to LVLMs, and generate the policy π directly. Actions in each ϕ_i are determined via its transition rule $\overline{\mathcal{T}}_i$. The robot executes low-level actions sequentially via parameterized primitive skills. Implementation details of the primitive skills used on the real robot are provided in Appendix G.

State Estimation and Model Updates. To ensure robustness against inaccuracies in generated appliance models, we adopt closed-loop model calibration. After each macro action is executed, the system gets new observations and tracks the state to check whether the target state is achieved. In simulation, the environment returns a textual description (e.g., "*cooking time is set to 30 min*"), while in the real world, an image is captured as visual feedback. In both cases, the feedback is passed to LVLMs to infer the actual value of the corresponding variables. If the actual value fails to match the expected one by the transition, ApBot first traverses the full value range of the variable

to observe how it responds to repeated actions. ApBot then utilizes the executed trace to update the transition $\overline{\mathcal{T}}$ (e.g., step size, value bounds), based on which the action sequence in this macro action $\mathbf{a}_\phi$ will be updated accordingly. With the updated macro actions, a new plan will be regenerated. For example, if `press("+")` produces a sequence like $0, \text{min} \rightarrow 10, \text{min} \rightarrow \cdots \rightarrow 60, \text{min} \rightarrow 0, \text{min}$, the updated rule reflects a 10-minute step and a wraparound at 60 minutes. The details of state tracking and model updates are elaborated in Appendix C.

4 Experiments

We evaluate ApBot by answering three questions: *(1) How does it compare to state-of-the-art LVLM agents for home appliance operation? (2) What are the main contributors of ApBot? (3) How does it perform on real-world appliances?* For (1), we compared ApBot with leading LVLM agents using unstructured inputs and found that ApBot consistently achieves higher success rates with fewer steps in both simulation and real-world settings. For (2), ablations show that structured appliance models, structured reasoning, and closed-loop updates are all critical for robust operation. For (3), real-world deployments demonstrate the effectiveness of ApBot on unseen appliances and long-horizon tasks.

4.1 Experimental Settings

Evaluation Benchmark. Our evaluation aims to systematically assess the effectiveness and generalization of ApBot across varying appliances. We construct a simulated benchmark of 30 interactive appliances with their manuals across 6 categories.

Task instructions are designed with varying numbers of variables, from simple to hard. Each instruction specifies explicit target values for the adjustable variables. In total, we evaluated each method on a set of 300 goal-directed natural language instructions, 10 per appliance instance. For automatic evaluation, each appliance in the benchmark is paired with a symbolic simulator that models true action effects and provides corresponding feedback to the algorithms. Full dataset including appliances image, user manual and instructions, along with the simulator details, is provided in Appendix D. For real-world evaluation, the system is deployed on three appliances using a Kinova Gen3 robot, following the same structured pipeline but relying on realistic visual observations for feedback. Fig. 7 shows the experimental setup.

Baselines. We compare ApBot with several baselines designed to ablate key components. *LLM as policy w/ image* uses LVLMs for all modules, including visual grounding [69] and reasoning based on unstructured, textual inputs. *LLM as policy w/ grounded actions* reasons over grounded actions from Sec. 3.3. We also conduct ablations as follows. *ApBot w/o model* does not build appliance model $\overline{\mathcal{M}}$. Instead, LVLMs (1) decide which action to execute directly; (2) if the LVLM deems that repeating steps is required, it invokes codes to get the required action sequences. *ApBot w/o button policy* builds a structured model $\overline{\mathcal{M}}$, and follows the macro actions in policy π strictly, but relies on LVLMs for low-level action generation instead of leveraging the transition $\overline{\mathcal{T}}$. *ApBot w/o close-loop update* disables model updates from observation feedback and executes in open-loop. Besides, we compare our action grounding approach with *Molmo*, a state-of-the-art visual grounding method. We elaborate all model settings and baselines in Appendix E.

Evaluation metrics. Success is defined as achieving all specified values correctly. For metrics, we evaluate (1) *Success Rate* within 25 reasoning steps, (2) *Average Steps* taken before success or termination, and (3) *Success weighted by Path Length (SPL)* to evaluate the weighted success rate considering the actual execution steps. Optimal steps are computed using oracle appliance models and task policies that specify the ground-truth action sequences.

4.2 Simulation Results

How does our framework compare to large-scale vision-language agents? The overall performance of ApBot is shown in Fig. 4. Compared to purely LVLM-based agents (*LLM as policy w/ image* and *LLM as policy w/ grounded actions*), ApBot achieves significantly better performance overall. Noticeably, comparing *LLM as policy w/ image* and *LLM as policy w/ grounded actions*, visually grounded actions overall help for appliance operation tasks. This shows that current state-of-

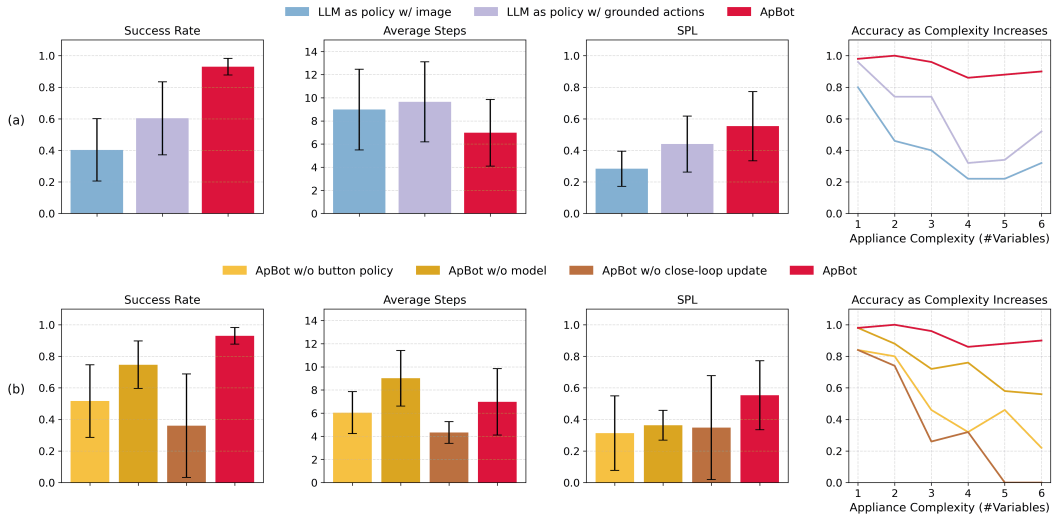


Figure 4: Overall performance of home appliance operation, including average task success rate (SR), average number of execution steps (Average Steps), and SPL (Success weighted by Path Length) across (a) baseline methods and (b) our ablations. Both performance and derivations are across appliance types.

the-art LVLMs are not yet good at open-vocabulary detection or visual grounding tasks, especially those requiring fine-grained text recognition. Detailed performance by appliance types is listed in Table 4. Referring to the rightmost figure in Fig. 4, we can see that ApBot has shown robustness against appliance complexity (from left to right). ApBot does not suffer from severe performance drop when the number of involved variables increases. By contrast, both baselines suffer from significant performance degradation. We also conducted χ^2 tests for all method pairs. As shown in Fig. 5, performance differences are statistically significant for all pairs except *ApBot w/o close-loop update* and *LLM as policy w/ image*, indicating they are equally poor. Detailed analysis by appliance type is provided in Appendix F.

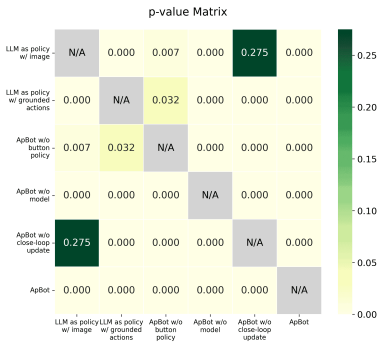


Figure 5: p-value matrix of all method pairs by χ^2-test.

What are the main contributors of ApBot? We conduct ablations to evaluate the contributions of key components in ApBot. In summary, removing the structured appliance models (*ApBot w/o model*) significantly degrades performance, mainly due to skipped steps or prematurely ending execution, which somehow mirrors the behavior of *LLM as policy w/ grounded actions*. This is because LVLMs cannot handle reasoning tasks involving a long history of many variables or constraints. It often ignores or hallucinates some of them (e.g., deciding whether the appliance is in the correct mode, proposing the required action to take), making the plan fail. Compared to *ApBot w/o button policy*, we can conclude that invoking code to compute required action sequences (Sec. 3.4) is crucial to ensure the correctness of generated policies. This is because LVLM agents struggle to assign variable values correctly when the variable range is large, when the transition $\mathcal{T}$ is complex or when the variable value options are semantically similar. Finally, we find that closed-loop updates for home appliance models are critical. Performance of *ApBot w/o close-loop update* suffers a rapid, sharp drop as the complexity of appliances increases. It fails to recover from any model errors, like open-loop policies. It reveals that current state-of-the-art LVLMs still struggle with generating constrained structures correctly in one shot, like the models of home appliances. All these results illustrate the necessity of structured reasoning for robust appliance operation. We further provide qualitative examples in Appendix F and failure analysis in Appendix H.

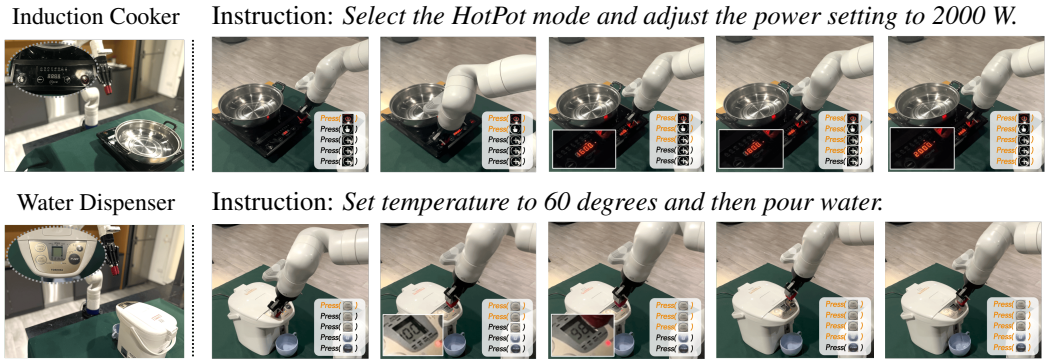


Figure 7: Snapshots of our system operating an induction cooker and a water dispenser.

What is the benefit of explicit action grounding? Our proposed method to ground actions can boost the overall performance of home appliance operation by 18% on average by comparison with the performance between *LLM as policy w/ image* and *LLM as policy w/ grounded actions* in Fig. 4, because LVLMs struggle with appliance button recognition. To further investigate the effectiveness of our action grounding methods, we tested and provided the visual grounding results for symbolic actions based on control panel images. The ground-truth labels of executable action regions are manually labeled. The comparison results between our method and *Molmo* are shown in Fig. 6. Our method is statistically significantly better than *Molmo* across all appliances (with p-value less than 0.001). The performance gain primarily comes from combining the advantages of (1) explicit text recognition, (2) high-recall detection, and (3) semantic understanding of graphical button icons of LVLMs. By contrast, *Molmo* demonstrates reasonable text or symbol recognition ability, yet not robust enough as the specialist OCR models.

4.3 Deployment on Real-Robot

We deploy our method on a Kinova Gen3 arm and demonstrate its applicability to three household appliances: a blender, an induction cooker, and a water dispenser, each evaluated with semantically diverse instructions. The button-pressing policy is parameterized by a bounding box. To compute the target end-effector pose, we compute the point cloud of the button and extract its surface normal. The robot aligns its gripper tip with the normal at a slightly tilted angle and moves 0.1 cm beyond the surface to ensure successful activation. More details can be founded in Appendix G. Fig. 7 illustrates two example instructions carried out on the water dispenser and the induction cooker, each frame executing an action. With our method, the robot can reason about how to perform previously unseen, long-horizon operation tasks by referring to the user manual. Additional real-world demonstrations and results can be found in Appendix G and the accompanying video.

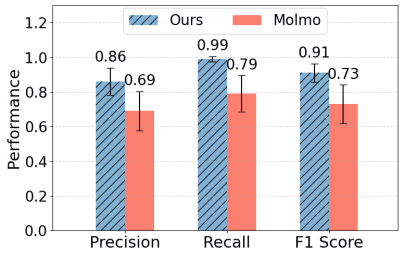


Figure 6: Comparison of action grounding performance between our method and Molmo. Standard deviation is across different appliance types.

5 Conclusion

We presented ApBot, a generalizable method that enables zero-shot operation of novel household appliances by referencing the user manual. By leveraging the structured model of appliances, ApBot demonstrates statistically significant robustness against diverse appliance types and language instructions. We built an evaluation suite including the benchmark of real-world appliances, manuals, open-ended task instructions, and symbolic simulators to benchmark home appliance operation. Compared to all baselines, ApBot significantly improves success rate. We also deployed and demonstrated our system on real-world robotic tasks. Results show that ApBot can reliably finish language-specified tasks autonomously with only the manual and visual observation as the inputs.

6 Failure Analysis & Limitations

Failure Analysis. Two major causes are identified in 18 failed instructions: action grounding (16.7%) and modeling (83.3%). Action grounding can fail when detectors misidentify soft-touch panels or icon-only buttons without physical boundaries. Modeling can fail due to goal ambiguity and persistent LLM hallucinations despite syntax checks. Failure details are in Appendix H.

Limitations. First, ApBot does not support touchscreens, which are becoming increasingly common. We will integrate material design of the manipulator to support such interfaces. Besides, button manipulation itself is a challenging task of robotics [15]. Currently, ApBot lacks fine-grained modelling of diverse buttons (e.g., frictions, tactile feedbacks), which are crucial for robust button manipulation. We will incorporate tactile sensing to improve reliability [70]. Also, the action grounding module is not fully reliable, especially for buttons without clear physical boundaries or with icon-only symbols. We will develop a robust detector to ground buttons or integrate a human-in-the-loop strategy [71]. Finally, ApBot does not consider complex manipulation skills for appliances, such as opening/closing doors, plugging, and putting in or removing items from the appliance container. We will integrate policy learning to support such sophisticated skills.

Acknowledgments

This work was supported by the NUS Research Scholarship.

References

[1] Y. Jiang, N. Walker, J. Hart, and P. Stone. Open-world reasoning for service robots. In *Proceedings of the international conference on automated planning and scheduling*, volume 29, pages 725–733, 2019.

[2] ManualsLib. Manuals library. https://www.manualslib.com/, 2025. Accessed April 22, 2025.

[3] Internet Archive. Digital library of free & borrowable texts, movies, music & wayback machine. https://archive.org/, 2025. Accessed April 22, 2025.

[4] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[5] M. Deitke, C. Clark, S. Lee, R. Tripathi, Y. Yang, J. S. Park, M. Salehi, N. Muennighoff, K. Lo, L. Soldaini, et al. Molmo and pixmo: Open weights and open data for state-of-the-art multimodal models. *arXiv preprint arXiv:2409.17146*, 2024.

[6] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[7] W.-J. Wang, C.-H. Huang, I.-H. Lai, and H.-C. Chen. A robot arm for pushing elevator buttons. In *Proceedings of SICE Annual Conference 2010*, pages 1844–1848. IEEE, 2010.

[8] A. A. Abdulla, H. Liu, N. Stoll, and K. Thurow. A robust method for elevator operation in semi-outdoor environment for mobile robot transportation system in life science laboratories. In *2016 IEEE 20th Jubilee International Conference on Intelligent Engineering Systems (INES)*, pages 45–50. IEEE, 2016.

[9] H. Nguyen, T. Deyle, M. Reynolds, and C. Kemp. Pps-tags: Physical, perceptual and semantic tags for autonomous mobile manipulation. In *Proceedings of the IROS Workshop on Semantic Perception for Mobile Manipulation*, 2009.

[10] D. Zhu, T. Li, D. Ho, T. Zhou, and M. Q. Meng. A novel ocr-rcnn for elevator button recognition. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3626–3631. IEEE, 2018.

[11] J. Liu, Y. Fang, D. Zhu, N. Ma, J. Pan, and M. Q.-H. Meng. A large-scale dataset for benchmarking elevator button segmentation and character recognition. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 14018–14024. IEEE, 2021.

[12] A. Yuguchi, T. Nakamura, M. Toyoda, M. Yamada, P. Tulathum, M. Aubert, G. A. Garcia Ricardez, J. Takamatsu, and T. Ogasawara. Toward robot-agnostic home appliance operation: a task execution framework using motion primitives, ontology, and gui. *Advanced Robotics*, 36 (11):548–565, 2022.

[13] N. Verzic, A. Chadaga, and J. Hart. Recovering missed detections in an elevator button segmentation task. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 13355–13362. IEEE, 2024.

[14] M. S. Sanders and E. J. McCormick. Human factors in engineering and design. *Industrial Robot: An International Journal*, 25(2):153–153, 1998.

[15] F. Wang, G. Chen, and K. Hauser. Robot button pressing in human environments. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 7173–7180. IEEE, 2018.

[16] V. Sukhoy and A. Stoytchev. Learning to detect the functional components of doorbell buttons using active exploration and multimodal correlation. In *2010 10th IEEE-RAS International Conference on Humanoid Robots*, pages 572–579. IEEE, 2010.

[17] A. Z. Ren, B. Govil, T.-Y. Yang, K. R. Narasimhan, and A. Majumdar. Leveraging language for accelerated learning of tool manipulation. In *Conference on Robot Learning*, pages 1531–1541. PMLR, 2023.

[18] S. Yang, O. Nachum, Y. Du, J. Wei, P. Abbeel, and D. Schuurmans. Foundation models for decision making: Problems, methods, and opportunities. *arXiv preprint arXiv:2303.04129*, 2023.

[19] J. X. Liu, Z. Yang, B. Schornstein, S. Liang, I. Idrees, S. Tellex, and A. Shah. Lang2ltl: Translating natural language commands to temporal specification with large language models. In *Workshop on Language and Robotics at CoRL 2022*, 2022.

[20] B. Liu, Y. Jiang, X. Zhang, Q. Liu, S. Zhang, J. Biswas, and P. Stone. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023.

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

[22] B. Vu, T. Migimatsu, and J. Bohg. Coast: Constraints and streams for task and motion planning. *arXiv preprint arXiv:2405.08572*, 2024.

[23] K. Lin, C. Agia, T. Migimatsu, M. Pavone, and J. Bohg. Text2motion: From natural language instructions to feasible plans. *Autonomous Robots*, 47(8):1345–1365, 2023.

[24] 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)*, pages 9493–9500. IEEE, 2023.

[25] 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)*, pages 11523–11530. IEEE, 2023.

[26] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

[27] Y. Yin, Z. Wang, Y. Sharma, D. Niu, T. Darrell, and R. Herzig. In-context learning enables robot action prediction in llms. *arXiv preprint arXiv:2410.12782*, 2024.

[28] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[29] S. Chen, A. Xiao, and D. Hsu. Llm-state: Expandable state representation for long-horizon task planning in the open world. *CoRR*, 2023.

[30] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36: 8634–8652, 2023.

[31] C. H. Song, J. Wu, C. Washington, B. M. Sadler, W.-L. Chao, and Y. Su. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 2998–3009, 2023.

[32] K. Nottingham, Y. Razeghi, K. Kim, J. Lanier, P. Baldi, R. Fox, and S. Singh. Selective perception: Optimizing state descriptions with reinforcement learning for language model actors. *arXiv preprint arXiv:2307.11922*, 2023.

[33] H. Jiang, B. Huang, R. Wu, Z. Li, S. Garg, H. Nayyeri, S. Wang, and Y. Li. Roboexp: Action-conditioned scene graph via interactive exploration for robotic manipulation. *arXiv preprint arXiv:2402.15487*, 2024.

[34] J. Ao, F. Wu, Y. Wu, A. Swikir, and S. Haddadin. Llm as bt-planner: Leveraging llms for behavior tree generation in robot task planning. *arXiv preprint arXiv:2409.10444*, 2024.

[35] H. Zhou, Y. Lin, L. Yan, J. Zhu, and H. Min. Llm-bt: Performing robotic adaptive tasks based on large language models and behavior trees. *arXiv preprint arXiv:2404.05134*, 2024.

[36] X. Chen, Y. Cai, Y. Mao, M. Li, W. Yang, W. Xu, and J. Wang. Integrating intent understanding and optimal behavior planning for behavior tree generation from human instructions. *arXiv preprint arXiv:2405.07474*, 2024.

[37] K. Rana, J. Haviland, S. Garg, J. Abou-Chakra, I. D. Reid, and N. Suenderhauf. Sayplan: Grounding large language models using 3d scene graphs for scalable task planning. *CoRR*, 2023.

[38] 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.

[39] 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.

[40] S. S. Raman, V. Cohen, I. Idrees, E. Rosen, R. Mooney, S. Tellex, and D. Paulius. Cape: Corrective actions from precondition errors using large language models. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 14070–14077. IEEE, 2024.

[41] S. Wang, M. Han, Z. Jiao, Z. Zhang, Y. N. Wu, S.-C. Zhu, and H. Liu. Llmˆ 3: Large language model-based task and motion planning with motion failure reasoning. *arXiv preprint arXiv:2403.11552*, 2024.

[42] Y. Long, J. Zhang, M. Pan, T. Wu, T. Kim, and H. Dong. Checkmanual: A new challenge and benchmark for manual-based appliance manipulation. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 22595–22604, 2025.

[43] S. Lin, A. Grastien, and P. Bercher. Towards automated modeling assistance: An efficient approach for repairing flawed planning domains. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 12022–12031, 2023.

[44] S. Lin, A. Grastien, and P. Bercher. Planning domain repair as a diagnosis problem. In *33rd International Workshop on Principle of Diagnosis–DX 2022*, 2022.

[45] S. S. Raman, V. Cohen, E. Rosen, I. Idrees, D. Paulius, and S. Tellex. Planning with large language models via corrective re-prompting. In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.

[46] W. Lu, R. K. Luu, and M. J. Buehler. Fine-tuning large language models for domain adaptation: Exploration of training strategies, scaling, model merging and synergistic capabilities. *arXiv preprint arXiv:2409.03444*, 2024.

[47] X. Zhang, Z. Altaweel, Y. Hayamizu, Y. Ding, S. Amiri, H. Yang, A. Kaminski, C. Esselink, and S. Zhang. Dkprompt: Domain knowledge prompting vision-language models for open-world planning. *arXiv preprint arXiv:2406.17659*, 2024.

[48] J. Zheng, H. Hong, X. Wang, J. Su, Y. Liang, and S. Wu. Fine-tuning large language models for domain-specific machine translation. *arXiv preprint arXiv:2402.15061*, 2024.

[49] B. Wang, Z. Wang, X. Wang, Y. Cao, R. A Saurous, and Y. Kim. Grammar prompting for domain-specific language generation with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[50] C. Tie, S. Sun, J. Zhu, Y. Liu, J. Guo, Y. Hu, H. Chen, J. Chen, R. Wu, and L. Shao. Manual2skill: Learning to read manuals and acquire robotic skills for furniture assembly using vision-language models. *arXiv preprint arXiv:2502.10090*, 2025.

[51] D. Nguyen, J. Chen, Y. Wang, G. Wu, N. Park, Z. Hu, H. Lyu, J. Wu, R. Aponte, Y. Xia, et al. Gui agents: A survey. *arXiv preprint arXiv:2412.13501*, 2024.

[52] S. Wang, W. Liu, J. Chen, Y. Zhou, W. Gan, X. Zeng, Y. Che, S. Yu, X. Hao, K. Shao, et al. Gui agents with foundation models: A comprehensive survey. *arXiv preprint arXiv:2411.04890*, 2024.

[53] V. Zhong, T. Rocktäschel, and E. Grefenstette. Rtfm: Generalising to novel environment dynamics via reading. *arXiv preprint arXiv:1910.08210*, 2019.

[54] A. W. Hanjie, V. Y. Zhong, and K. Narasimhan. Grounding language to entities and dynamics for generalization in reinforcement learning. In *International Conference on Machine Learning*, pages 4051–4062. PMLR, 2021.

[55] G. Li and Y. Li. Spotlight: Mobile ui understanding using vision-language models with a focus. In *The Eleventh International Conference on Learning Representations*, 2023.

[56] K. Cheng, Q. Sun, Y. Chu, F. Xu, Y. Li, J. Zhang, and Z. Wu. Seeclick: Harnessing gui grounding for advanced visual gui agents. *arXiv preprint arXiv:2401.10935*, 2024.

[57] Z. Wu, Z. Wu, F. Xu, Y. Wang, Q. Sun, C. Jia, K. Cheng, Z. Ding, L. Chen, P. P. Liang, et al. Os-atlas: A foundation action model for generalist gui agents. *arXiv preprint arXiv:2410.23218*, 2024.

[58] B. Gou, R. Wang, B. Zheng, Y. Xie, C. Chang, Y. Shu, H. Sun, and Y. Su. Navigating the digital world as humans do: Universal visual grounding for gui agents. *arXiv preprint arXiv:2410.05243*, 2024.

[59] Y. Lu, J. Yang, Y. Shen, and A. Awadallah. Omniparser for pure vision based gui agent. *arXiv preprint arXiv:2408.00203*, 2024.

[60] H. He, W. Yao, K. Ma, W. Yu, Y. Dai, H. Zhang, Z. Lan, and D. Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6864–6890, 2024.

[61] J. Y. Koh, R. Lo, L. Jang, V. Duvvur, M. Lim, P.-Y. Huang, G. Neubig, S. Zhou, R. Salakhutdinov, and D. Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 881–905, 2024.

[62] Y. Qin, Y. Ye, J. Fang, H. Wang, S. Liang, S. Tian, J. Zhang, J. Li, Y. Li, S. Huang, et al. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

[63] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[64] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM (JACM)*, 30(2):323–342, 1983.

[65] 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*, 33:1877–1901, 2020.

[66] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, et al. Segment anything. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4015–4026, 2023.

[67] M. Minderer, A. Gritsenko, and N. Houlsby. Scaling open-vocabulary object detection. *Advances in Neural Information Processing Systems*, 36:72983–73007, 2023.

[68] JaidedAI. EasyOCR: Ready-to-use OCR with 80+ supported languages. https://github.com/JaidedAI/EasyOCR, 2020.

[69] J. Yang, H. Zhang, F. Li, X. Zou, C. Li, and J. Gao. Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v. *arXiv preprint arXiv:2310.11441*, 2023.

[70] S. Yu, K. Lin, A. Xiao, J. Duan, and H. Soh. Octopi: Object property reasoning with large tactile-language models. In *Robotics: Science and Systems (RSS)*, 2024.

[71] A. Xiao, N. Janaka, T. Hu, A. Gupta, K. Li, C. Yu, and D. Hsu. Robi butler: Remote multimodal interactions with household robot assistant. *arXiv preprint arXiv:2409.20548*, 2024.

A Example of the Appliance Model and Simulation

A.1 Structured Appliance Model Example

Below is an example of the appliance model generated using ApBot for a *dehumidifier*. It includes a list of variables extracted from the manual, the macro actions, and transitions. During inference, we directly generate the model in the following format with the help of LVLM agents, based on which we further generate operation plans with Prompt 8. Note that the *macro action*, which is typically a concept in computer science, is phrased as *feature* in our prompts to match the commonly used term in most of the manuals. In practice, we group consecutive actions of adjusting the same variable in a macro action into a *step*, which empirically improves robustness and facilitates the syntax checking (Sec. B).

```
# Variables of the appliance defined in the State
variable_power_on_off = DiscreteVariable(value_range=["on", "off"],
↪  current_value="off")
variable_fan_speed = DiscreteVariable(value_range=["low", "mid",
↪  "high"], current_value="low")
...

# Macro actions
feature_list = {}

feature_list["turn_on_off"] = [
    {"step": 1, "actions": ["press_power_button"], "variable":
    ↪  "variable_power_on_off", "step_size": 2}
]
feature_list["adjust_fan_speed"] = [
    {"step": 1, "actions": ["press_speed_button"], "variable":
    ↪  "variable_fan_speed", "step_size": 3}
]
...

# Transitions
simulator_feature = Feature(feature_list=feature_list,
↪  current_value=("empty", 1))

class Simulator(Appliance):

    def reset(self):
        self.feature = simulator_feature
        self.variable_power_on_off = variable_power_on_off
        self.variable_fan_speed = variable_fan_speed
        ...

    def press_power_button(self):
        self.feature.update_progress("press_power_button")
        self.execute_action_and_set_next("press_power_button")

    def press_speed_button(self):
        self.feature.update_progress("press_speed_button")
        self.execute_action_and_set_next("press_speed_button")

    ...
```

B Details of the Syntax Checker

To mitigate hallucination during appliance model generation, we implement a suite of syntax checkers to further validate the generated models, mainly for the macro actions, transitions, and goal specifications. Additionally, all generated codes are verified to ensure that they fit the required output format, with the help of *regular expressions*. The detailed prompt can be found in Prompt 6. We list the syntax checkers here:

1. **Missing Variable:**

   Every step should adjust some variables.

2. **Empty or Non-Existent Action:** Each step should contain at least one valid action.

3. **Action Coverage:** Every action in $\overline{A}$ should appear in some macro actions.

4. **Variable Coverage.** Every variable defined in the state space $\overline{S}$ should appear in some macro actions.

5. **Duplicate Action Sequences:** We check if there are possibly duplicate action sequences (e.g., set a variable to a specified value twice).

6. **Number-Pad Action Compatibility:** Number-pad actions should not appear when modeling appliances without a number pad.

7. **Input String Reset:** The appliance with a number pad should reset the input string of the number pad whenever it switches away.

8. **Action-Variable Consistency:** Actions should only adjust associated variables.

9. **Goal Validity:** $\overline{S}_g$ should be fully specified, i.e., each variable should be assigned or intentionally ignored.

C Details of State Estimation and Model Updates

State Estimation. The robot estimates the appliance state using two feedback modalities. In simulation, textual feedback directly provides ground truth values for state variables being tuned. In real-world scenarios, the robot captures an image of the appliance and uses LVLM agents to convert visual observations into textual state descriptions. After completing each macro action, the robot compares the predicted state resulting from the planned action with the observed state extracted from feedback. The result indicates whether the macro action successfully achieved its intended effect.

Model Updates. The generated operation plan is executed in the minimal unit of a macro action. The robot does not track states or update appliance models during the execution of a macro action, but only after its completion. If the observed state does not match the predicted state, it indicates that some transitions in this macro action might be wrong. Hence, the robot initiates a sequence of exploration actions to explore and fix the possible errors. Empirically, we found that go-to transitions in $\overline{\mathcal{T}}_g$ are mostly correct. Therefore, for each action in $\overline{A}_n$, the robot continuously executes it until a previously observed state is observed again. This exploration strategy is based on the observation that most actions in $\overline{A}_n$ for appliances are circular. Based on the observed state transitions, the robot updates the transition model regarding the corresponding action and regenerates all macro actions that depend on it.

An Example of State Estimation and Model Updates. Below is an example illustrating how the appliance model is automatically updated using closed-loop feedback. Consider a task that requires turning on the fan and setting the fan speed to `high`. The robot begins by executing the `press_power_button` action under the macro action of `turn_on_off`. Upon receiving feedback indicating `power = on`, it invokes Prompt 9 to confirm that the subgoal is achieved as expected, i.e., the prediction matches the observation. Next, it proceeds to the macro action of `adjust_fan_speed`. Assuming the current speed is `low`, the robot executes the action sequence in this macro action.

Figure 8: Appliances in our benchmark. (a) Appliance Types. (b) All Instances of Bread Maker.

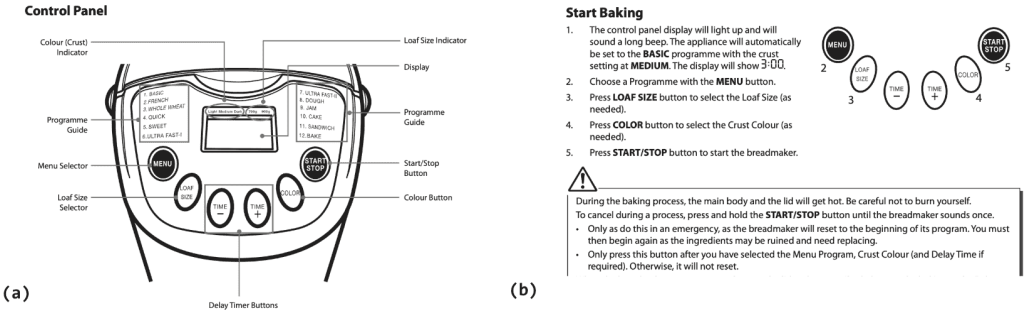


Figure 9: An example user manual for the Bread Maker. (a) Control panel. (b) Unstructured step-by-step procedure for a macro action.

Given a ground-truth cyclic variable range of {`low`, `medium`, `high`}, assume that the current action sequence is wrong, for example, requiring 1 press of the speed button, which results in `medium` speed. After executing the planned actions, the feedback again indicates `speed = medium`, suggesting the goal is not yet met. The robot continues pressing the speed button and observes the feedback sequence: `medium` → `high` → `low` → `medium`. The repeated value `medium` indicates the entire value range has been cycled. Using Prompt 10, the robot diagnoses that the transition of the action `adjust_fan_speed` was incorrect. It then updates the model according to the diagnosis. In detail, it sequentially updates the variable definition via Prompt 11, revises the appliance model using Prompt 12, and adjusts the goal state accordingly using Prompt 13. With the updated current value of `medium`, the robot re-plans the action sequence and presses the speed button once more. This time, the feedback confirms `speed = high`, and Prompt 9 verifies that the goal is satisfied. The task completes successfully.

D Details of the Evaluation Benchmark

D.1 Appliances Categories and Data Collection

The benchmark covers six types of household appliances: *dehumidifiers*, *bottle washers*, *rice cookers*, *microwave ovens*, *bread makers*, and *washing machines*. These categories were chosen for their differences in mechanism complexity and functional diversity. As shown in Figure 8, each category includes five distinct instances, resulting in 30 appliances in total.

For each instance, we collect an image of the control panel from the Internet, including Amazon and eBay. We also collect the corresponding user manual from the product's official website or support page. From each manual, we extract two key parts: (1) a control panel legend that links interface elements to their locations (see Fig. 9a), and (2) step-by-step instructions that describe how to operate specific features (see Fig. 9b). They are used to construct the structured symbolic model for each appliance. The appliances vary in interface layout and the number of adjustable variables. To

Table 1: Examples of Task Instructions for Different Appliances

# Vars	Appliance	Sample Instruction	Target Settings
1	Dehumidifier	"Set the humidity to 50%."	• Humidity = 50%
2	Bottle Washer	"Power on the device and initiate a 45-minute automatic sterilization and drying cycle."	• Power = On • Drying Time = 45 min
3	Rice Cooker	"Adjust the delay timer to 30 minutes, set the rice cooker to White Rice mode, and start the operation."	• Menu = White Rice • Delay Timer = 30 min • Start = On
4	Microwave Oven	"Set the upper tube temperature to 150°C. Select the cooking function as 'upper and lower heating tube'. Then set the lower tube temperature to 150°C and adjust the cooking time to 20 minutes."	• Upper Temp = 150°C • Time = 20 min • Lower Temp = 150°C • Function = Upper / Lower Heating
5	Bread Maker	"Bake a large, medium-crust French loaf using the French menu. Set a 2-hour delay timer, then start the bread maker."	• Menu = French • Loaf Size = Large • Crust = Medium • Delay = 2 hrs • Start = On
6	Washing Machine	"Turn on the washing machine. Select the Normal program for everyday clothes, set the water level to 55 L, schedule it to finish in 4 hours, start the machine, and activate the child lock."	• Power = On • Program = Normal • Water Level = 55 L • Preset = 4 hrs • Start = On • Child Lock = On

ensure fair comparison, we assign a fixed number of target variables per appliance type, subject to their inherent complexity. For example, *dehumidifiers* require adjusting one variable, while washing machines require six.

D.2 Task Instructions

We design 300 goal-directed natural language instructions, with 10 tasks per appliance instance. Each instruction specifies a clear goal by assigning specific target values to a set of variables. This ensures consistency across methods and focuses evaluation on symbolic reasoning and execution. Ground-truth values are manually labeled to support automatic evaluation. The number of variables involved in each task depends on the type of appliance, facilitating controllable comparison. For example, all instructions for *dehumidifier* involve only one variable. More complex ones, like washing machines, involve up to six variables. Details and samples of instructions are listed in Table 1.

D.3 Simulators and Ground Truth Feedback

Each appliance instance is paired with a symbolic simulator implemented in Python, providing a deterministic testing environment. The simulator encodes adjustable variables and valid actions based on the appliance manual, preserving constraints defined by its manual. It also defines executable regions tied to control panel elements. Actions are input as a pair: a bounding box and an action type (e.g., press or turn). If the action is valid, the simulator updates the variable state and returns a textual message (e.g., `temperature = 150°C`) indicating the resulting variable value.

D.4 Metrics

We mainly evaluate *Success Rate*, *Average Step*, and *Success weighted by Path Length*. Besides, we also report the *Execution Step* in Appendix F.

Success Rate: It is defined as the proportion of tasks completed successfully before exceeding 25 *reasoning steps*. The number of reasoning steps is defined as the total number of macro actions, excluding the exploration steps. We define *success* as the achievement of variable values included in the specified goal state at the end of execution.

Average Step: This metric indicates the average number of *reasoning steps*, i.e., the number of macro actions excluding the exploration, taken before either achieving success or reaching the maximum number (25 in our experiments). This metric focuses on the reasoning efficiency of the system.

Success weighted by Path Length (SPL): SPL evaluates success while considering the actual number of physical *execution steps*, i.e., symbolic actions in $\overline{A}$. The optimal number of actions is

Table 2: Hyperparameters of Used Models.

Component	Parameter Name	Explanation	Value
GPT	`model`	GPT model version.	`GPT-4o-2024-11-20`
	`temperature`	Controls output randomness.	`1.0`
	`top_p`	Nucleus sampling cutoff.	`1`
OWLv2	`model`	OWLv2 model name.	`owlv2-large-patch14-ensemble`
	`box_threshold`	Object detection threshold.	`0.5`
EasyOCR	`text_threshold`	Text detection threshold.	`0.5`
	`low_text`	Includes blurry text.	`0.4`
	`contrast_ths`	Contrast enhancement threshold.	`0.05`
Segment Anything Model	`iou`	IoU threshold for masks.	`0.1`
	`conf`	Mask confidence filter.	`0.9`

manually labeled by a human oracle. Intuitively, this metric evaluates the efficiency considering both execution and exploration.

Execution Step: It indicates the average number of symbolic actions in $\overline{A}$, i.e., the number of actions that the robot actually executes physically, including the exploration ones, taken before success or reaching the max reasoning steps. It evaluates the physical execution efficiency of algorithms.

E Details of Experimental Settings

E.1 Detailed Settings of LVLMs

Table 2 lists non-default hyperparameters of all models used in our experiments. GPT-4o was used for both appliance model construction and action grounding. Claude-3.5, EasyOCR, and OWLv2 were used only for action grounding. In real-world experiments, where control panels are simpler, only OWLv2 was used for control element detection to improve efficiency; EasyOCR and SAM were omitted.

E.2 Baselines

Table 3 summarizes the key components in each method. ✓indicates the component is present in the corresponding method; × indicates it is omitted or replaced directly by LVLM equivalents. The prompts used for all methods can be found in Appendix I.

Grounded Action refers to whether the method reasons over symbolic actions, which are visually grounded to control panel elements via our action grounding method (see Sec. 3.3), or directly reasons over image regions without symbolic abstraction.

Model $\overline{\mathcal{M}}$ indicates whether the method constructs and follows a symbolic appliance model extracted from the user manual. If present, variable adjustments follow a fixed sequence specified by macro actions $\overline{\Phi}$ and the task policy π, instead of being chosen reactively by an LLM.

Button Policy denotes whether action sequences for variable adjustment are computed using predefined transition functions $\overline{\mathcal{T}}$, rather than being generated by LLMs.

Closed-loop Update refers to whether the method incorporates execution feedback to update state estimation and re-generate action sequences. Methods without this component operate in an open-loop manner, executing fixed sequences or reasoning reactively without correcting for execution errors.

Table 3: Baseline methods and ablation of key components in ApBot. ✓indicates the component is used.

Method	**Grounded Action**	**Model $\overline{\mathcal{M}}$**	**Button Policy**	**Closed-loop Update**
LLM as policy w/ image	×	×	×	✓
LLM as policy w/ grounded actions	✓	×	×	✓
ApBot w/o model	✓	×	✓	✓
ApBot w/o button policy	✓	✓	×	✓
ApBot w/o close-loop update	✓	✓	✓	×
ApBot	✓	✓	✓	✓

F Details of Performance

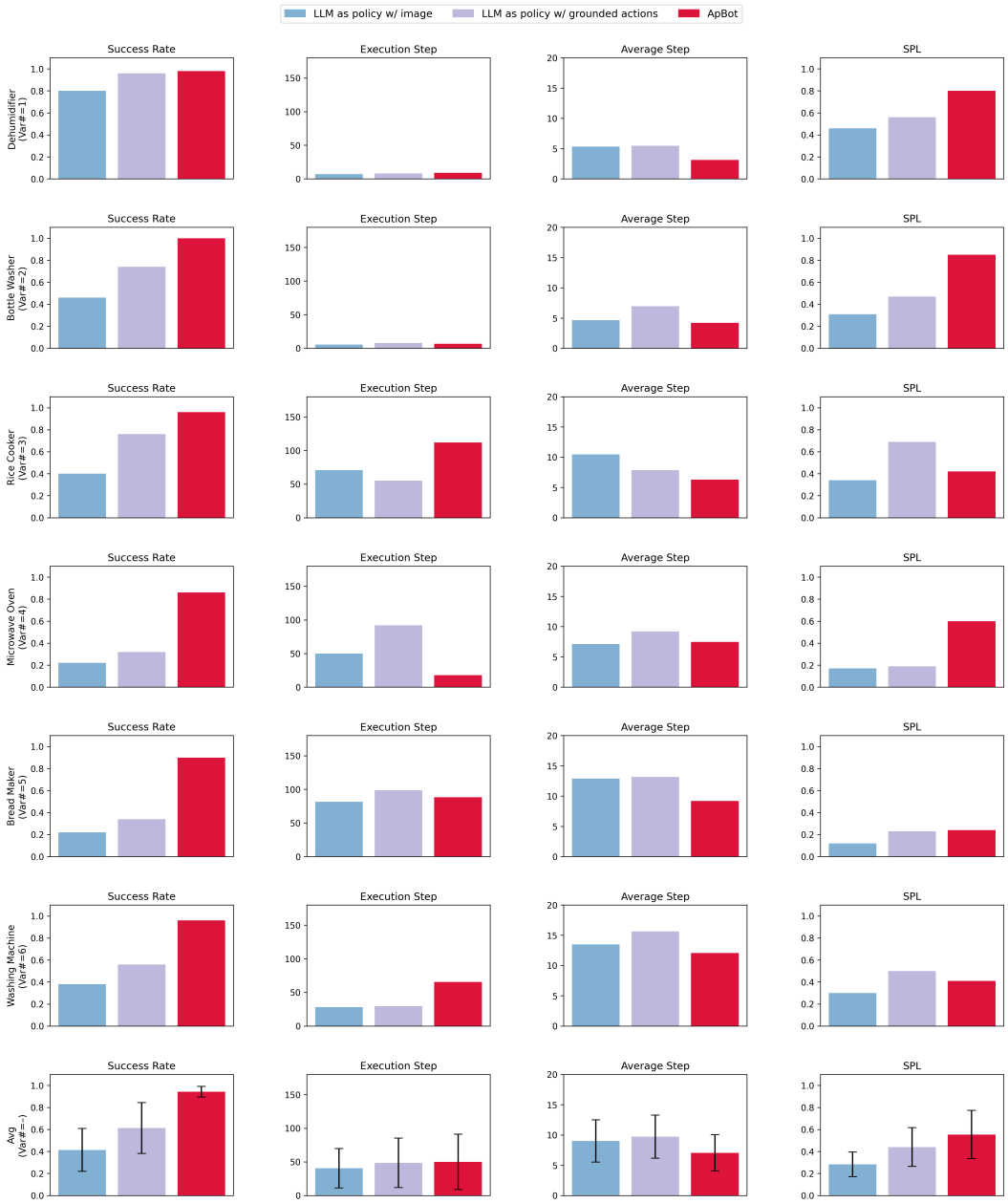


Figure 10: Performance of home appliance operation by appliance type, including average task success rate (SR), average number of execution steps (Average Steps), and SPL (Success weighted by Path Length) on baseline methods.

Figure 10 and Figure 11 shows the performance of ApBot on six appliance types. Each appliance type has a different number of variables to adjust, from 1 to 6 (top to bottom). As the number of variables increases, ApBot does not suffer a severe performance drop in terms of success rate (Figure 13). By contrast, baseline methods like *LLM as policy w/ image* and *LLM as policy w/ grounded actions* drop significantly on tasks with more variables. This shows that structured models, structured reasoning, and closed-loop updates help in handling complex tasks. Another interesting observation is that SPL suffers from an obvious drop when increasing the complexity of appliances. The reason is that for complex appliances and tasks, there will always be more modeling errors,

Figure 11: Performance of home appliance operation by appliance type, including average task success rate (SR), average number of execution steps (Average Steps), and SPL (Success weighted by Path Length) on ablation methods.

which require more exploration steps for model updates. This further demonstrates the necessity of appliance modeling and online updates.

Figure 12 presents pairwise χ^2-test p-values across six methods for each appliance type, with diagonal entries marked as "N/A". Each subplot corresponds to an appliance type, ordered by the number of variables to adjust per user instruction. As the number of variables increases, the performance gap between ApBot and baseline methods such as *LLM as policy w/ image* and *LLM as policy w/ grounded actions* becomes more statistically significant.

Figure 14 compares action grounding performance between ApBot and Molmo across six appliance types, evaluated by precision, recall, and F1 score. ApBot consistently outperforms Molmo on all

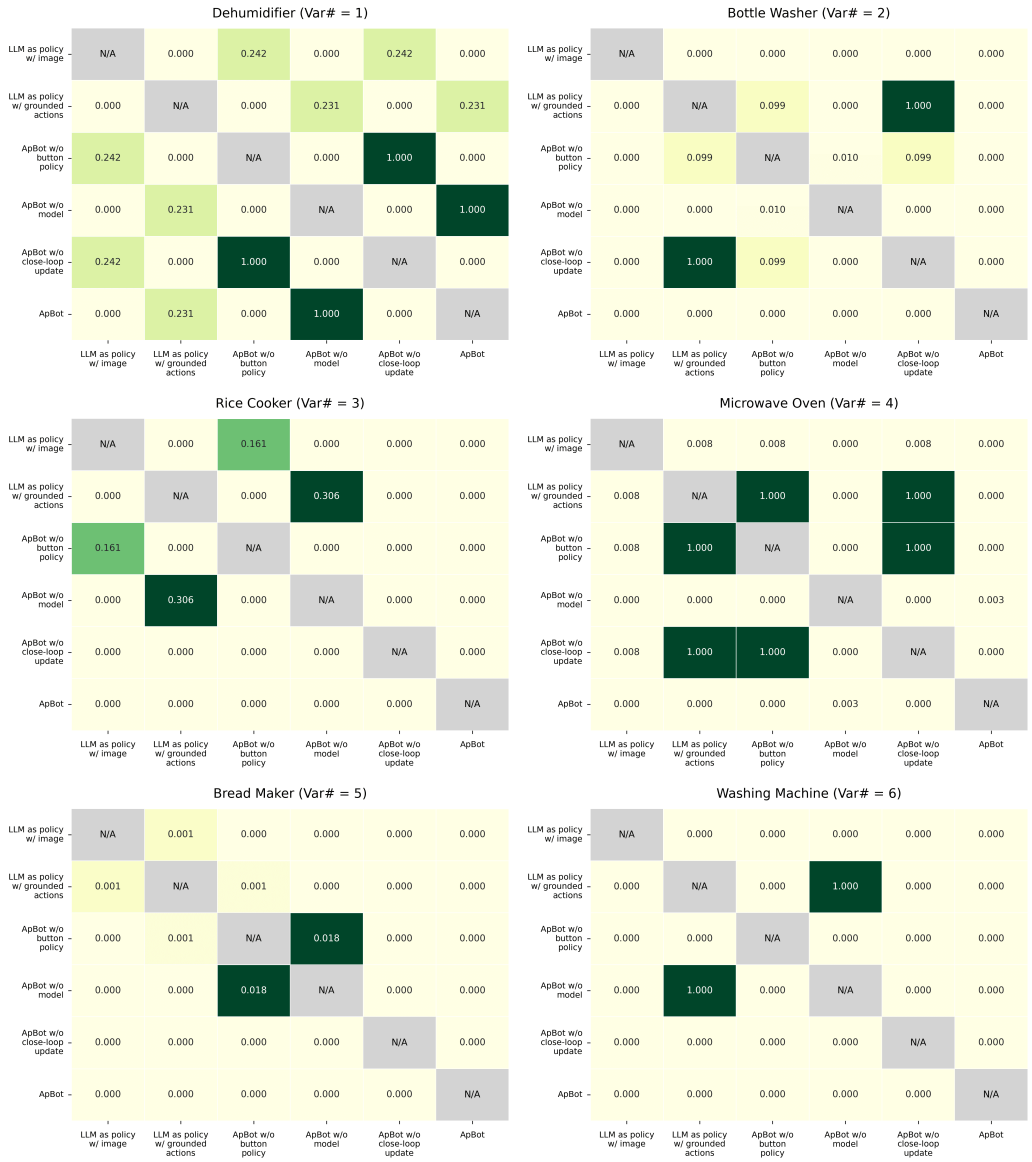


Figure 12: p-value matrix of all method pairs by χ^2-test on different appliance types.

appliance types, particularly on appliances with symbolic, iconic, or multi-word text labels, where a structured grounding procedure performs better.

We illustrate an online model update example triggered by a transition failure. An incorrect transition rule for the microwave function dial extracted from the user manual led to a goal mismatch. Upon observing inconsistent state feedback, ApBot exhaustively explores the function dial's state space and updates the macro action to reflect the correct transition mapping.

Update Macro Action: Adjust Microwave Function

Action Applied: ('turn_function_dial_clockwise', 1)

Feedback Received:

- Fermentation

Goal Comparison:

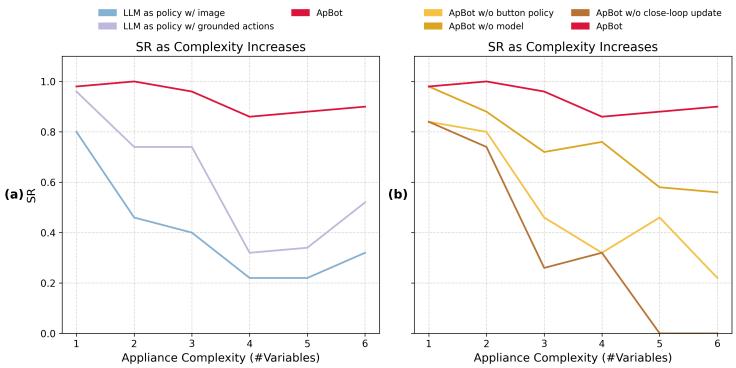


Figure 13: Average task success rate (SR) by increasing variable size conditioned on appliance type.

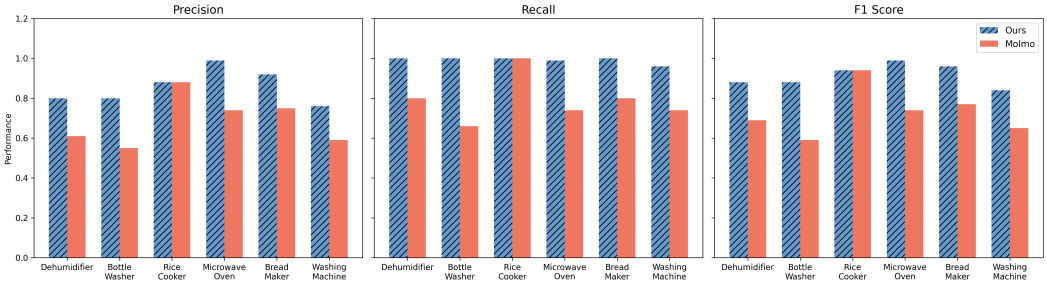


Figure 14: Comparison of action grounding performance between our method and Molmo on precision and recall across appliance types.

- **Expected:** Lower & Upper Heater
- **Observed:** Fermentation
- **Result:** Values are semantically different. Goal not reached.

Execution Trace:

Action	Observed Value
(‘turn_function_dial_clockwise’, 1)	Fermentation
(‘turn_function_dial_clockwise’, 1)	Lower heater
(‘turn_function_dial_clockwise’, 1)	Upper heater
(‘turn_function_dial_clockwise’, 1)	Lower & upper heater
(‘turn_function_dial_clockwise’, 1)	Convection
(‘turn_function_dial_clockwise’, 1)	Rotary
(‘turn_function_dial_clockwise’, 1)	Off
(‘turn_function_dial_clockwise’, 1)	Fermentation
(‘turn_function_dial_clockwise’, 1)	Lower heater

Inferred Variable Definition:

- **Name:** `variable_function`
- **Type:** `DiscreteVariable`
- **Value Range:**
  `['Fermentation', 'Lower heater', 'Upper heater', 'Lower & upper heater', 'Convection', 'Rotary', 'Off']`
- **Current Value:** `'Off'`

Generated Code:

```
variable_function_knob = DiscreteVariable(
    value_range=[
```

Table 4: Detailed Performance of Success Rate / Average Steps by Appliance Types

Method	Dehumid-ifier	Bottle washer	Rice cooker	Microwave oven	Bread maker	Washing machine
LLM as policy w/ image	0.80 / 5.34	0.46 / 4.64	0.40 / 10.46	0.22 / 7.12	0.22 / 12.88	0.32 / 13.46
LLM as policy w/ grounded actions	0.96 / 5.48	0.74 / 6.94	0.74 / 7.88	0.32 / 9.22	0.34 / 13.02	0.52 / 15.36
ApBot w/o button policy	0.84 / 3.28	0.80 / 4.58	0.46 / 5.22	0.32 / 6.98	0.46 / 8.18	0.22 / 8.04
ApBot w/o model	0.98 / 5.32	0.88 / 7.22	0.72 / 7.80	0.76 / 10.38	0.58 / 11.92	0.56 / 11.44
ApBot w/o close-loop update	0.84 / 2.82	0.74 / 3.82	0.26 / 4.34	0.32 / 4.00	0.00 / 5.34	0.00 / 5.62
Ours	0.98 / 3.12	1.00 / 4.22	0.96 / 6.28	0.86 / 7.46	0.88 / 9.14	0.90 / 11.64

```
        'Fermentation', 'Lower heater', 'Upper heater',
        'Lower & upper heater', 'Convection', 'Rotary', 'Off'
    ],
    current_value='Off'
)
```

G Details of Real World System

G.1 Real World System Design

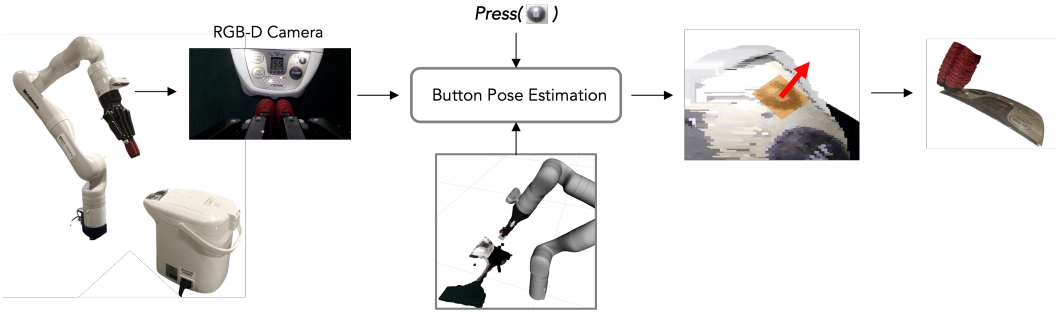


Figure 15: The real-world framework

In our real-world robotic system, we implement a framework that enables a manipulator to interact with physical appliances by pressing buttons accurately and robustly, as illustrated in Figure 15. An RGB-D camera mounted near the robot's end-effector captures both RGB images and depth data of the appliance interface. Given a press action parameterized by the bounding box of the target button, the button pose estimation module extracts the corresponding point cloud and computes the surface normal of the button region. This normal vector determines the correct approach angle for the robot to align its end-effector.

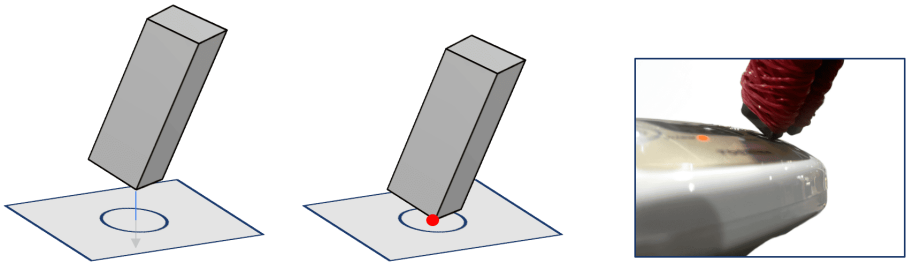


Figure 16: The pressing details

To reduce the contact area and improve precision, the robot aligns its gripper with the surface normal at a slight tilt. The pressing trajectory is generated in two stages: first, the end-effector moves to a position directly above the button; then, it advances 0.1 cm beyond the estimated button surface to ensure a firm press, as shown in Figure 16. This approach compensates for minor depth inaccuracies and mechanical backlash, enhancing contact reliability. The generated trajectory is executed using workspace tracking control, allowing the end-effector to follow the desired pressing motion precisely. This framework generalizes well across various devices and button types, demonstrating robustness to differences in button size, orientation, and mechanical resistance.

G.2 Real World Experiments Setting

To evaluate the performance and generalization ability of our proposed system, we conducted a series of real-world experiments involving common household appliances. Specifically, the robot was tasked with operating three distinct devices: a blender, an induction cooker, and a water dispenser, as shown in Fig. 17. These appliances were selected for their diversity in interface design and physical interaction requirements, representing different types of button layouts, activation mechanisms, and task objectives. For each appliance, we designed three task scenarios, resulting in a total of nine distinct interaction tasks. These tasks involve activating power buttons, selecting modes (e.g., milk mode or hot pot mode), or dispensing liquids, depending on the appliance. These tasks require the system to generalize based on visual input and prior knowledge for reasoning encoded in the user manual.

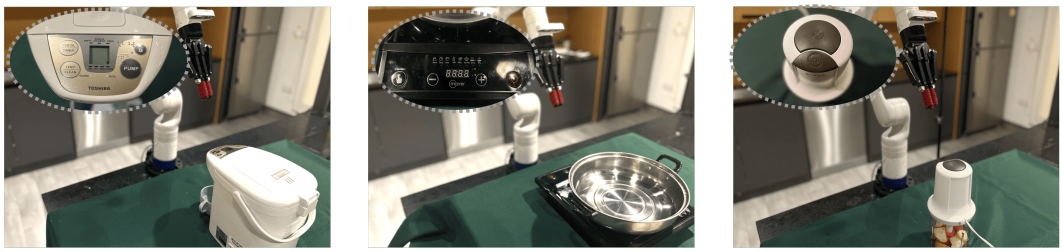


Figure 17: The real world setting

Table 5: Detailed Performance of Success Rate / Execution Step / SPL by Appliance Types

Method	Blender	Water Dispenser	Induction Cooker	Avg
LLM as policy w/ image	1.0 / 3.50 / 0.29	1.0 / 5.70 / 0.59	0.5 / 17.70 / 0.13	0.83 / 8.97 / 0.34
ApBot	1.0 / 1.0 / 1.0	0.7 / 8.4 / 0.31	1.0 / 16.5 / 0.38	0.9 / 8.63 / 0.56

T1. *Select the HotPot mode and set power to 2000 W.*
T2. *Select the Milk mode.*
T3. *Select the HotPot mode and set power to 1600 W.*
T4. *Set the insulation temperature to 98°, then pour the water.*
T5. *Set the insulation temperature to 85°, then pour the water.*
T6. *Set the insulation temperature to 65°, then pour the water.*
T7. *Hold at slow speed for 10 seconds.*
T8. *Hold at slow speed for 15 seconds.*
T9. *Hold at turbo speed for 10 seconds.*

G.3 More Real World Execution Visualization

To evaluate the impact of parsing visual feedback from appliance displays, we built simulators for these three appliances, each using images of digital control panels to reflect state changes. For each appliance, 10 instructions were tested. LVLM agents were used to parsed the display after each action to infer the updated state, which was passed back as feedback to guide the next step. For LLM as policy w/ image, the image was directly given to LVLM agents. For ApBot, feedback parsing is done by guiding the LVLM to focus on the variable currently being adjusted (Prompt 18). We compare LLM as policy w/ image and ApBot in terms of success rate, execution steps, and SPL, as shown in Table 5. Due to the simplicity of these appliances, both methods show similar performance. For more visualization, please see Fig. 18.

H Details of Failure Mode Analysis

We categorize and analyze the main failure modes observed across different baseline methods as shown below.

Failure due to Lack of Action Grounding This is defined as failures that occur due to incorrect grounding of actions to visual elements. For example, the model may select a neighboring button instead of the correct one because LVLMs struggle to associate OCR text labels with the correct control panel element. This highlights limitations in visual-text alignment within vision-language models. This failure mode is mainly applicable to: LLM as policy w/ image.

Failure due to Lack of Structured Model This mainly include failures that occur due to (1) incorrect association between actions and effects due to lack of transition modeling; (2) repeated adjustment of the same variable due to lack of macro actions; (3) premature ending or wrongly parsed visual feedback due to lack of state estimation; (4) incorrect goal state specification due to

lack of structured goal states. This failure mode is mainly applicable to LLM as policy w/ image; LLM as policy w/ grounded actions; ApBot w/o model.

Failure due to Incorrect Transition Model This mainly includes failures caused by incorrect interpretation of transition rules, especially when the variable exhibits irregular step sizes in its value space. This failure mode is mainly applicable to: ApBot w/o button policy; ApBot w/o close-loop update.

Failure due to Hallucinated Model Details This includes failures caused by LLM hallucination during model construction, resulting in invalid transition rules that fail syntax checks. This failure mode is mainly applicable to: ApBot, ApBot w/o button policy, ApBot w/o model, ApBot w/o close-loop update.

I Prompts

In this section, we provide the detailed prompts for two baselines: LLM as policy w/ image (Prompt 1) and LLM as policy w/ grounded actions (Prompt 2). The remaining ablation methods share the same prompts as ApBot.

For ApBot, we provide prompts for three sections: (1) Build appliance models; (2) Update appliance models using closed-loop feedback; (3) Action grounding. To build appliance models, we need to (1) extract control panel element names (Prompt 3) and action names (Prompt 4); (2) extract variables (Prompt 5), macro actions (Prompt 6), and generate the appliance model with extracted information (Prompt 7); finally (3) Generate task policy and goal state based on the appliance model (Prompt 8).

To update the appliance model using closed-loop feedback, the steps include: (1) After execution of each macro action and receiving feedback, ApBot parses the feedback (Prompt 18) and compares the goal with the feedback (Prompt 9). If the goal is achieved, it proceeds to the next action. Otherwise, it executes exploration actions to collect a sequence of observations. Then, it uses them to diagnose the incorrectly modeled variable (Prompt 10), updates the variable definition (Prompt 11), and updates the appliance model (Prompt 12) and goal state (Prompt 13) accordingly.

To perform action grounding, we need to: (1) Use LVLMs to detect candidate bounding boxes for control panel elements, then remove false positives using LVLMs (Prompt 14). This step ensures only valid regions are kept before passing them for slower, more detailed grounding. (2) Map bounding boxes to control panel element names (Prompt 15). (3) Remove duplicate bounding boxes being mapped to the same control panel element (Prompt 16). And (4) Map each action name to a grounded control panel element name and an action type (Prompt 17).

Prompt 1: LLM as policy w/ image Action Proposal

You are given:

- Two images:
  (1) A photo of the appliance control panel.
  (2) A version with indexed bounding boxes circling the control panel elements (buttons, dials).
- A user command describing the target task.
- User manual.
- A set of allowed action types: `press`, `hold`, `turn_dial_clockwise`, `turn_dial_anti_clockwise`.
- Optionally, display panel feedback in text after each action.

Action Proposal Rules:

- At the start of the task, assume the initial appliance state is unknown. Execute an action to receive feedback. On subsequent steps, use observed display panel feedback to reason about the current state, and propose the next action needed to complete the task.
- Only one action is allowed per response, but you can execute it multiple times (e.g., set `execution_times = 2`.

- `hold` actions require specifying a duration. If not mentioned in manual, default to 10 seconds. `hold` can involve two buttons simultaneously and requires a `duration`. The other action types apply to a single button or dial.
- If the task is completed or infeasible (e.g., display feedback remains wrong after repeated attempts failed), return an `end` action to stop.

Output Format: Return 5 Python variables in the following format:

```
variable_reason = "<Your reasoning>"
action_type = "press_button"  # or other valid type
bbox_index = 5                 # int or [int, int] if pressing two
↪  buttons
execution_times = 1           # integer count
duration = None               # duration in seconds if hold, otherwise
↪  None

# to terminate a task:
variable_reason = "Task is completed / unable to achieve."
action_type = "end"
bbox_index = None
execution_times = None
duration = None
```

Example:

```
# User instruction: Set the dial (index = 8) from \texttt{OFF} to
↪  \texttt{3}.
variable_reason = "Current power value is OFF. I will turn the dial
↪  clockwise 3 times to set it to 3."
action_type = "turn_dial_clockwise"
bbox_index = 8
execution_times = 3
duration = None
```

Prompt 2: LLM as policy w/ grounded actions Action Proposal

You are given:

- A user command describing the target task.
- User manual.
- A list of available executable actions.
- Optionally, display panel feedback in text after each action.

Action Proposal Rules:

- At the start of the task, assume initial appliance state is unknown. Execute an action to receive feedback. On subsequent steps, use observed display panel feedback to reason about current state, and propose the next action needed to complete the task.
- Use only the listed available actions. Each action should be returned as a Python function call. Provide a clear and concise reason using `variable_reason`.
- Only one action is allowed per response, but you can execute it multiple times (e.g., set `execution_times = 3`).
- If a `hold` action causes values to change too quickly, avoid using it. Use repeated `press` actions instead. `hold` actions require specifying a duration. If not mentioned in the manual, default to 10 seconds.
- If the task is completed or infeasible (e.g., display feedback remains incorrect after repeated attempts), return an `end` action to stop.

Output Format: Return 2 Python variables in the following format:

```
variable_reason = "<Your reasoning>"
variable_response_string = "run_action('action_name',
↪  execution_times=N)"

# Example of hold actions:
variable_response_string = "run_action('hold_buttonX_and_buttonY',
↪  execution_times=1, duration=5)" # 5 seconds

# To terminate the task:
variable_reason = "Task is completed / unable to achieve."
variable_response_string = "end"
```

Example:

```
# User instruction: Set the dial from OFF to 3 by turning it
↪  clockwise.
variable_reason = "Current power value is OFF. I will turn the dial
↪  clockwise 3 times to set it to 3."
variable_response_string = "run_action('turn_dial_clockwise',
↪  execution_times=3)"
```

Prompt 3: Extract Control Panel Element Names

You are given an appliance user manual and an image of its control panel. Identify all **control panel elements**, i.e., `button` and `dial`.

Identification Guidelines:

- Include elements mentioned in the manual or shown in the image if they clearly correspond to a described function.
- Use one name per physical control. If it adjusts multiple settings, use a combined name (e.g., `power_timer_dial`, not `power_dial` and `timer_button`). If the manual names a button (e.g., `function_button`), use that name, even if the image shows only labels of its configurations like `menu 1`, `menu 2`, `menu 3`.
- List each distinct button separately, even if they adjust the same function. Examples: `air_roast_button`, `air_fry_button`; `increase_button`, `decrease_button`; `number_0_button`, `number_1_button`, ...

Exclude:

- Non-executable parts such as printed labels, static icons, light indicators, and digital displays.
- Any component not on the control panel, such as power plugs or lids.

Naming Conventions:

- Use `name_type` format (e.g., `start_stop_button`, `power_level_dial`).
- Only lowercase letters, digits, and underscores are allowed. No spaces or special characters.

Output Format:

- Return a Python list named `names_list`.
- Each item must be a string with a Python comment describing its function, location, and any visible symbol (e.g., triangle, bottle, arrow).

Example Output:

```
names_list = [
    "start_stop_button",  # starts/stops cooking; lower right;
    ↪  triangle icon
    "number_1_button",    # sets time; middle keypad; labeled '1'
    "increase_button",    # increases value; top left; '+' symbol
]
```

Prompt 4: Extract Action Names

You are given an appliance user manual and a list of **control panel element** names. Your task is to identify all **executable actions** that are:

(1) **described in the user manual**, and
(2) Involve **control panel elements** listed above (e.g., buttons, dials).

Carefully match each control element with relevant actions described in the manual.

Valid Action Types:

- `press_<element_name>`
- `hold_<element_name>` #(duration = x seconds; use 3 if unspecified)
- `hold_<element1>_and_<element2>` #(duration = x seconds; use 3 if unspecified)
- `turn_<element_name>_clockwise` (only valid for dials)
- `turn_<element_name>_anticlockwise` (only valid for dials)

Naming conventions:

- Construct each action by selecting a valid action type from the list above and inserting a control element name from the provided list.
- Use lowercase letters, digits, and underscores only. Do not include any special characters or symbols.

Exclusions:

- Do not include actions not mentioned in the manual.
- Do not create duplicate or ambiguous actions.
- Do not include duration in the action name. Write it as a comment on the same line.

Output Format: List each valid action as a separate line of plain text.

Example Output:

```
press_kitchen_timer_button
press_time_dial
press_and_hold_stop_button  #(duration = 5 seconds)
press_and_hold_start_button_and_cancel_button  #(duration = 3 seconds)
turn_power_level_dial_clockwise
turn_power_level_dial_anticlockwise
```

Prompt 5: Extract Variables

You are given an appliance user manual, a list of executable action names, a list of control panel element names, and a list of predefined variable classes in Python. Your task is to extract all appliance **variables** as instances of the predefined Python classes.

Definition of Variable: An internal configuration state of the appliance that can be adjusted through actions (e.g., power level, temperature, time).

How to Identify a Variable: User manuals often describe multiple **features** (i.e. high-level functions like `Defrost`, `Grill`), each consisting of actions that configure internal appliance states. These states are the **variables**. For example, a microwave may include `Defrost` and `Grill` features, both of which adjust `menu` and `time`, but assign different values depending on the feature. Here, `Defrost` and `Grill` are features. `menu` and `time` are variables shared across features. Define a variable if:

(1) It is explicitly **described in the manual**,
(2) It is adjusted via a **listed control panel element name** (e.g., button, dial), and
(3) It is modified by an **listed action action**.

Naming Convention: Use the format `variable_<variable_name>`. Use only lowercase letters and underscores.

```
variable_power_on_off = ... # User manual: Press POWER to turn off.
variable_child_lock = ...
variable_start_pause = ...
```

Valid Variable Types: Used to define variable transition rules. Each variable type can be directly invoked via code. Each variable can have its value changed by `.next()` and `.prev()` or directly assigned by `.set_current_value()`.

(1) `DiscreteVariable`: Categorical values. Value range consists of strings.

```
variable_power = DiscreteVariable(value_range=["on", "off"],
↪  current_value="on")
variable_mode = DiscreteVariable(value_range=["eco", "turbo",
↪  "auto"], current_value="eco")
```

(2) `ContinuousVariable`: Numerical values. Supports piecewise ranges.

```
variable_clock_setting_hour =
↪  ContinuousVariable(value_ranges_steps=[[0, 23, 1]],
↪  current_value=0) # value range: 0-23 hours, step size: 1
↪  hour
variable_wash_time = ContinuousVariable(value_ranges_steps=[[0,
↪  3, 3], [3, 15, 1]], current_value=0) # value range: 0 or
↪  3-15 minutes
```

(3) `TimeVariable`. Supports "hour-minute-second" format.

```
variable_timer = TimeVariable(values_ranges_steps =
↪  [('00:00:00', '00:59:00', 60)], current_value='00:00:00') #
↪  value range: 0-59 minutes: step size: 1 min
```

(4) `InputString`. Stores keypad input sequence.

```
# User manual: Enter a 3-digit code using number pads to set
↪  the timer.
variable_input_string = InputString()
```

Output Format: Executable python code that defines each variable. The current variable value should be initialised to the first value in the range if not otherwise specified by the manual.

Example Output:

```
variable_power = DiscreteVariable(value_range = ["on", "off],
↪  current_value = "on")
variable_temperature = ContinuousVariable(value_ranges_stpes = [[20,
↪  30, 1]], current_value = 20)
```

Special Cases:

(1) **Setting Adjustable via Different Features:** If a setting can be adjusted in different features using different transition rules (i.e. how a variable's value changes given an action), define a separate variable for each (e.g., `cook_time` set via number pads vs. incremented by `press_start_button`).

```
# User manual <normal cook>:
# 1) Press "COOK" once;
# ...
# 4) Use the number pads to enter cooking time in MM:SS format
↪  (e.g., to set 6 minutes, press "6", "0", "0");
# 5) Press "COOK" again to confirm.
variable_normal_cook_time = ...

# User manual <speedy cook>:
# 1) Press "Start" to start cooking for 30 seconds. Each
↪  subsequent press adds time by 30 seconds.
variable_speedy_cook_time = ...
```

(2) **Setting Adjusted Across Different Feature Steps:** If a setting is adjusted in multiple steps (e.g., hour and minute of a timer) in a feature, define one variable per step.

```
# User manual <clock setting>:
# 1) Press "CLOCK" once, the hour figure flashes.
# 2) Press "up arrow" or "down arrow" to adjust the hour
↪  (0--23).
# 3) Press "CLOCK", the minute figure flashes.
# 4) Press "up arrow" or "down arrow" to adjust the minute
↪  (0--59).
# 5) Press " CLOCK " to finish clock setting. ":" will flash,
↪  the "clock symbol" indicator will go out. The clock setting
↪  has been finished.
variable_clock_setting_hour = ...
variable_clock_setting_minute = ...
```

(3) **Setting Conditioned on Program Choice:** If a setting's value range depends on the selected program, (e.g. microwave menu, washing machine program), follow this structure.

- Define a selector variable, e.g., `variable_program_index`, to store the chosen program.
- Define a placeholder variable, e.g., `variable_program_setting = None`, which is dynamically assigned.
- For each program, define a separate variable using the format `variable_<feature_name>_<program_name>` (e.g., `variable_set_program_popcorn`).
- Create a dictionary `program_setting_dict` to map each program to its respective setting variable.

```
# User manual:
# Microwave program popcorn sets size (1 cup, 2 cup), pizza
↪  sets weight (250g, 350g, 450g), soup sets volume (200ml,
↪  300ml, 400ml).
# Each time a new program is selected, variable_program_setting
↪  is updated using program_setting_dict.

# variable A (selector)
variable_program_index = DiscreteVariable(["popcorn", "pizza",
↪  "soup"], "popcorn")

# variable B (placeholder)
variable_program_setting = None

# program-specific variables
variable_program_setting_popcorn = DiscreteVariable(["1 cup",
↪  "2 cup"], "1 cup")
variable_program_setting_pizza = DiscreteVariable(["250g",
↪  "350g", "450g"], "250g")
variable_program_setting_soup = DiscreteVariable(["200ml",
↪  "300ml", "400ml"], "200ml")

# mapping dictionary
program_setting_dict = {
    "popcorn": variable_program_setting_popcorn,
    "pizza": variable_program_setting_pizza,
    "soup": variable_program_setting_soup
}
# Selecting a mode updates variable_menu_setting from this
↪  dictionary.
```

Prompt 6: Extract Features

You are given the user manual of an appliance, a list of executable action names, a list of variables, and a predefined `Feature()` class in Python. Your task is to extract all appliance **features** as an instance of the predefined `Feature()` object.

Definition of Feature: A high-level operation (e.g., `clock setting`, `cooking`) consisting of step-by-step procedures that adjust one or more variables using valid actions.

Output Format: Define a dictionary `feature_list`, where each item is a feature name and its value is a list of steps. Each step is a dictionary with:

(1) `step` index (integer),
(2) `actions` (list of action strings),
(3) Optional `variable` adjusted in this step,
(4) Optional `comment` describing fixed action effects or input string parsing requirements.

If any actions or variables are unused, include them under the reserved feature `"null"`:

```
feature_list["null"] = [{"step": 1, "actions": ["unused_action_1"],
↪  "missing_variables": ["variable_a"]}]
```

Conclude with:

```
simulator_feature = Feature(feature_list=feature_list,
↪  current_value=("empty", 1))
```

Example Output:

```
# User manual <clock setting>:
# 1) Press "CLOCK" once, the hour figure flashes.
# 2) Press "up arrow" or "down arrow" to adjust the hour (0--23).
# 3) Press "CLOCK", the minute figure flashes.
# 4) Press "up arrow" or "down arrow" to adjust the minute (0--59).
# 5) Press " CLOCK " to finish clock setting. ":" will flash, the
↪  "clock symbol" indicator will go out. The clock setting has been
↪  finished.

feature_list = {}
feature_list["clock_setting"] = [
    {"step": 1, "actions": ["press_clock_button"]},
    {"step": 2, "actions": ["press_up_arrow_button",
    ↪  "press_down_arrow_button"], "variable":
    ↪  "variable_clock_setting_hour"},
    {"step": 3, "actions": ["press_clock_button"]},
    {"step": 4, "actions": ["press_up_arrow_button",
    ↪  "press_down_arrow_button"], "variable":
    ↪  "variable_clock_setting_minute"},
    {"step": 5, "actions": ["press_clock_button"]}
]
feature_list["null"] = [{"step": 1, "actions": [],
"missing_variables": []}]
simulator_feature = Feature(feature_list=feature_list,
↪  current_value=("empty", 1))
```

Identification Guidelines:

(1) Only model features with clear step-by-step instructions written in the user manual. Ignore features introduced only by naming buttons and dials without full procedures.

(2) Exclude non-essential features like WiFi, app control, remote control, reset, cleaning, multi-stage cooking, sound/audio settings, memory, touchscreen feedback, or progress queries after operation starts. For `hold_<element>` actions, ignore action effects that merely speed up changes. Only model a `hold` action if it toggles a function (e.g., child lock).

(3) Split features into shorter, reusable units where possible. For *consecutive steps in a feature*, if they adjust different variables, consider separating them into distinct features (e.g. `start`, `cancel`, `power_on`). If *consecutive steps in a feature* adjust the same variable (e.g., `lock`/`unlock`), merge them.

(4) The feature that should stay merged is program settings, as the specific program setting is conditioned the program choice (e.g. `pizza` program requires setting `cooking_weight`, but `soup` program requires setting `soup_volume` (explained in *extract variable*). Follow this structure:

```
feature_list["set_program"] = [
  {"step": 1, "actions": ["press_program_button"], "variable":
  ↪  "variable_program_index"},
  {"step": 2, "actions": ["press_plus_button",
  ↪  "press_minus_button"], "variable":
  ↪  "variable_program_setting"}
]
```

(5) If an action always sets a variable to a fixed value, remark in `"comment"`.

```
feature_list["start_cooking"] = {"step": 1, "actions":
↪  ["press_start_button"], "variable":
↪  "variable_start_cooking",
 "comment": "start always set to on"}
```

(6) If an action affects multiple variables, set the variable whose values will be assigned dynamically under `variable` and describe those with fixed target values in `comment`.

```
  # user manual <speedy cooking>:
  # press start button will immediately start cooking at 100%
  ↪  power for 30 seconds. Each subsequent press increases
  ↪  cooking time by 30 seconds.
feature_list["start_cooking"] = {"step": 1, "actions":
↪  ["press_start_button"], "variable":
↪  "variable_cooking_time",
 "comment": "variable_start set to on, variable_power set to
 ↪  100"}
```

(7) `turn_dial` actions must match both direction and effect. If `turn_dial` affects different variables in different directions, distinguish them (e.g., clockwise for `time`, anticlockwise for `power`).

```
feature_list["adjust_time"] = [{"step": 1, "actions":
↪  ["turn_dial_clockwise"], "variable": "variable_time"}]
feature_list["adjust_power"] = [{"step": 1, "actions":
↪  ["turn_dial_anticlockwise"], "variable": "variable_power"}]
```

(8) To compactly describe appliance features that input values via number pads, you can use the given `meata_actions_on_numbers` to refer all the number pads, and track them with `meta_actions_dict`. Make a comment beside the variable whose value assignment requires parsing from input string.

```
# Predefined
meta_actions_on_number = [
    "press_number_0_button", "press_number_1_button", ...,
    ↪  "press_number_9_button"
]
meta_actions_dict = {
    "0": "press_number_0_button",
    "1": "press_number_1_button",
    ...
}

# Example usage
feature["set_timer"] = [
{"step": 1, "actions": ["press_timer_button"],},
{"step": 2, "actions": meta_actions_on_numbers, "variable":
↪  "variable_timer",
 "comment": "requires parsing from variable_input_string"}]
```

Prompt 7: Extract Appliance Model

You are given a user manual, a list of action names, variables, features, and a predefined `Appliance()` class in Python. Your task is to implement a **Simulator()** object as an instance of the predefined `Appliance()` object that models all action effects of the appliance.

Definition: The `Simulator()` object inherits from `Appliance()` and implements three components:

(1) `reset()` method that assigns:
- `self.feature`, initialized as `simulator_feature`.
- `self.variable_x`, initialized from predefined variables.
- `self.variable_input_string`, `self.meta_actions_dict`, etc., if appliance includes number pads.

(2) Action functions that define effects on variables and features. Valid action effects include:

- Advance the current feature step or switch features by calling `self.feature.update_progress(action_name)`. Current feature and step index can be accessed by `self.feature.current_value`.
- Get active variable via `self.get_current_variable(action_name)`.
- Conditionally update variable value ranges or step size.
- Update variable value with `variable_x.set_current_value()`, `self.assign_variable_to_next(variable_x)`, or `self.assign_variable_to_prev(variable_x)`.

```
def press_a_button(self):
    self.feature.update_progress("press_a_button")
    current_feature = self.feature.current_value[0]
    variable = self.get_current_variable(action_name)
    if current_feature == "feature_a":
        variable.set_current_value("on")
    elif current_feature in ["feature_b", "feature_c"]:
        self.assign_variable_to_next(variable)
```

(3) `run_action(action_name, ...)` is a wrapper that enforces global execution conditions before running an action. Specifically:

- Prevents action execution when the appliance is locked or powered off, unless the action is to unlock or power on.
- Clears the input buffer (i.e., `self.variable_input_string`) if the action is unrelated to input via number pads.
- After passing precondition checks, invokes the corresponding action method to perform its effect.

```
def run_action(self, action_name, execution_times=1, **kwargs):
    if action_name not in self.meta_actions_dict.values():
        self.variable_input_string.input_string = ""
    if self.variable_lock.get_current_value() == "locked" and
    ↪  "unlock" not in action_name:
        self.display = "child lock: locked"
        return self.display
    return super().run_action(action_name, execution_times,
    ↪  **kwargs)
```

Example Output:

```
class Simulator(Appliance):

    def reset(self):
        self.feature = simulator_feature
        self.variable_clock_setting_hour = variable_clock_setting_hour
        self.variable_clock_setting_minute =
        ↪  variable_clock_setting_minute

    def press_clock_button(self):
        ...

    def press_up_arrow_button(self):
        ...

    def press_down_arrow_button(self):
        ...

    def run_action(self, action_name, execution_times=1, **kwargs):
        ...
```

Other Valid Action Function Formats:

(1) For `hold_<element_name>` actions, the duration needs to be included.

```
def press_and_hold_lock_button(self, duration=3):
    if duration >= 3:
        self.feature.update_progress("press_and_hold_lock_butt
        ↪  on")
        ...
```

(2) If the action changes a program choice (e.g. `microwave menu`, `washing machine program`), sometimes the available program settings will change (e.g. `pizza` program requires setting `cooking_weight`, but `soup` program requires setting `soup_volume` (explained in *extract variable*). Update the `variable_program_setting` accordingly.

```
def press_menu_button(self):
    ...
    self.variable_program_setting = self.program_setting_dict[
    ↪  self.variable_program_index.get_current_value()]
```

(3) If an action involves pressing number pads, follow this structure.

- Define a `press_number_button` method to model number pad action effects. Use this method to instantiate specific number pad actions.

```
# number pad action effects.
def press_number_button(self, action_name, digit):
    self.feature.update_progress(action_name)
    self.variable_input_string.add_digit(digit)
    variable = self.get_current_variable(action_name)
    value = self.process_input_string(current_feature,
    ↪  variable_name)
    variable.set_current_value(value)

# instantiate specific number pad actions.
def press_number_2_button(self):
    self.press_number_button("press_number_2_button", "2")
```

- Define a `process_input_string` to convert inputs via number pads (e.g. `"1"`, `"6"`, `"0"`, `"0"`) to valid variable values (e.g. clock time of `"16:00"`).

```
# converts time inputs of minute:second format to
↪  hour:minute:second format
def process_input_string(self, feature, variable_name):
    raw_input = self.variable_input_string.input_string
    if feature == "clock_setting" and variable_name ==
    ↪  "variable_clock_time":
        time_string = "00" + str(raw_input).zfill(4)
        return f"{time_string[:2]}:{time_string[2:4]}:{ti
        ↪  me_string[4:]}"
```

- Define a `get_original_input` to convert target variable values (e.g. clock time of `"16:00"`) to required inputs via number pads (e.g. `"1"`, `"6"`, `"0"`, `"0"`).

```
# converts target time value of hour:minute:second format
↪  to required inputs of minute:second format
def get_original_input(self, goal, feature, variable_name):
    digits_only = ''.join(char for char in str(goal) if
    ↪  char.isdigit())
    if feature == "clock_setting" and variable_name ==
    ↪  "variable_clock_time":
        return digits_only[2:].lstrip("0") or "0"
```

- In `reset()` method, add the following content.

```
def reset(self):
    ... (the aforementioned variable assignments)
    self.variable_input_string = VariableInputString()
    self.meta_actions_dict = meta_actions_dict
    self.meta_actions_on_number =
    ↪  self.meta_actions_on_number
```

Prompt 8: Generate Task Policy and Goal State

You are given a user manual, a list of features, a list of variables, and a user instruction. Your task is to determine which features need to be executed and how variables should be set to fulfill the instruction.

Output Formats

(1) a Python list `task_policy` which defines the minimal ordered list of features needed to fulfill the user instruction. Use the following rules:
- Every selected feature must set at least one variable required in the user instruction.
- Exclude features whose variables are all covered by previous features.
- Include the feature to turn on the device and let it start running.

(2) a string `policy_choice_reason` that explains why each feature was selected. If multiple features are needed, explain what each contributes.

(3) a `changing_variables` list that includes all variables in the feature sequence, in order of appearance. Only include listed variables.

(4) a `goal_state = Simulator()` object. For each variable in `changing_variables`, assign its target value following this structure:
- Use `set_current_value()` for direct assignment.
- Use `set_value_range()` or `set_step_value()` if the variable's default configuration changes.
- Do not modify variable names. Use the exact names from `changing_variables`.
- For `ContinuousVariable` and `TimeVariable`, add a Python comment indicating unit (e.g., seconds, minutes, hours).

Example Output:

```
# User Instruction: Defrost chicken meat for 5 minutes at 50% power in
↪ 3 hours time.
task_policy = ["cook", "preset", "start"]
policy_choice_reason = "Firstly adjust cook settings then set preset
↪ hours."
changing_variables = ["variable_microwave_cooking_power",
↪ "variable_microwave_cooking_time", "variable_preset_time",
↪ "variable_start"]
goal_state = Simulator()
goal_state.variable_microwave_cooking_power.set_current_value("P50")
goal_state.variable_microwave_cooking_time.set_current_value("00:05:0
↪ 0") # 5
↪ minutes
goal_state.variable_preset_time.set_current_value(3)  # hour
goal_state.variable_start.set_current_value("on")
```

Handle Program Choices: An appliance may allow choosing different programs (e.g. microwave menu, washing machine program), and each program has different settings (e.g. `pizza` program requires setting `cooking_weight`, but `soup` program requires setting `soup_volume` (explained in *extract variable*). In this case, `variable_program_setting` will be initialized with `None` in `reset()`. Therefore in `goal_state`, firstly assign it to an existing defined variable (e.g., from a mapping dictionary), and set its value accordingly.

```
# Given variables
variable_program_index = DiscreteVariable(["popcorn", "pizza",
"soup"], "popcorn"),
variable_program_setting = None

variable_program_setting_popcorn = DiscreteVariable(["1 cup",
"2 cup"], "1 cup"),
variable_program_setting_pizza = DiscreteVariable(["250g",
"350g", "450g"], "250g"),
variable_program_setting_soup = DiscreteVariable(["200ml",
"300ml", "400ml"], "200ml"),

program_setting_dict = {
"popcorn": variable_program_setting_popcorn,
"pizza": variable_program_setting_pizza,
"soup": variable_program_setting_soup
}

# Given feature
feature_list["set_program"] = [
    {"step": 1, "actions": ["press_program_button"], "variable":
    ↪  "variable_program_index"},
    {"step": 2, "actions": ["press_up_arrow_button",
    ↪  "press_down_arrow_button"], "variable":
    ↪  "variable_program_setting"}
]

# User Instruction: Set the microwave to cook 1 cup of popcorn...
task_policy = ["set_program"]
policy_choice_reason = "This feature contains variable_program_index
↪  and variable_program_setting".
changing_variables = ["variable_program_index",
↪  "variable_program_setting"]
goal_state = Simulator()
goal_state.variable_program_index.set_current_value("popcorn")
goal_state.variable_program_setting = variable_program_setting_popcorn
goal_state.variable_program_setting.set_current_value("1 cup")
```

Prompt 9: Compare Goal State with Feedback

You are given the appliance model, together with two strings in the format `variable_name: variable_value`, representing the goal state and the real-world feedback, respectively. Your task is to determine whether the feedback indicates the goal is reached.

Comparison Rules:

(1) Allow equivalent variable-value meaning. *E.g.* `variable_menu = "Popcorn"` vs. `mode_popcorn = "on"` ⇒ `True`; `variable_power = "On"` vs. `variable_on_off = "On"` ⇒ `True`

(2) If values contain both numbers and text, remove text and compare numbers. Ignore casing or formatting if numerically identical. *E.g.* `"0g"` vs. `"0"` ⇒ `True`; `"100cm"` vs. `"100"` ⇒ `True`; `"1 cup"` vs. `"1 serving"` ⇒ `True`

(3) Ensure the match is the closest in the value range. *E.g.* `program="wash"` v.s `program="wash, dry"`, both values exist in value range ⇒ `False`

Output Format:

- `reason`: a string explaining your judgment.
- `goal_reached`: either `True` or `False`.

Example Output:

```
# goal: popcorn setting = 100g;
# feedback: popcorn: 100
reason = "Both values represent 100g, ignoring unit suffix."
goal_reached = True
```

Prompt 10: Diagnose Incorrect Variable Definition

You are given:

- A list of defined variable names in the appliance model.
- A variable name `variable_x` suspected to be incorrectly defined.
- A full step-by-step execution record starting from the first observed change in that variable's value. Each record includes the action taken and the observed result in the format: `variable_name = variable_value`.

Your Tasks:

(1) **Identify the root variable:**
- Match the observed variable name to the closest name in the given variable list. If the mismatch is caused by this variable itself, return that name as `variable_name`.
- If the variable is conditioned on a program choice (e.g., `variable_program_setting`), and the mismatch is due to a sub-variable (e.g., `variable_program_setting_popcorn`), return the name of the sub-variable.

(2) **Determine if the variable is continuous:**
- Return `variable_is_continuous = True` if the values are numeric and increase/decrease regularly.
- Else return `variable_is_continuous = False`.

(3) **Extract the variable values as a list:**
- Extract all values of the observed variable in order from the record.
- Store them in `record_sequence`.
- Use `int/float` for continuous variables and `str` for discrete ones.

Output Format: Return the following Python variables:

- `variable_name`
- `variable_is_continuous`
- `record_sequence`

Example:

```
# inputs given
defined_variables = [
    "variable_wash_time",
    "variable_spin_speed",
    "variable_temperature"
]

execution_record = [
  {step_index: 1, action: ("turn_dial", 1), observation: wash_time =
  ↪  6},
  {step_index: 2, action: ("turn_dial", 1), observation: wash_time =
  ↪  9},
  {step_index: 3, action: ("turn_dial", 1), observation: wash_time =
  ↪  12},
  ...
]

# Expected Output
variable_name = "variable_wash_time"
variable_is_continuous = True
record_sequence = [6, 9, 12, ...]
```

Prompt 11: Update Variable Definition from Observed Values

You are given the following inputs:

- `variable_name`: the variable that has been confirmed to be incorrectly defined.
- `variable_is_continuous`: whether the variable is continuous or discrete.
- `record_sequence`: the list of observed values of the variable over time.
- The current implementation of the variable.
- The user manual and a guide for valid variable definitions.

Your Task: Update the variable definition by modifying its current value, value range, step size, or value order to match all values in `record_sequence`.

Instructions:

(1) **Paste the reasoning trace:** Insert the provided `record_sequence` as Python comments to justify your updates.

(2) **Update the variable:** Modify the definition of the chosen variable to match observed behavior. Keep the same name. Valid modifications include:
   (a) **Change variable type** according to observation.
   (b) **Change current value** to match with the last observed value.
   (c) **Adjust value range or step size** if the record shows regular repetition. Use piecewise ranges if steps skip sections.
   (d) **Change value order** for discrete variables if observed cycling order differs.

(3) **Copy related data structures:** If the variable is part of a program-conditioned setting (e.g., `variable_program_setting`, explained in *extract variables*), also update the program dictionary:

```
program_setting_dict["menu_x"] = variable_x
```

(4) **Align with real-world units.** For example, if feedback is in `cm`, don't define value ranges in `m`. For continuous variables representing time or weight, indicate the unit in a Python comment (e.g., seconds, minutes, grams).

Example Output:

```
# given inputs
variable_name = "variable_program_setting_popcorn"
variable_is_continuous = True
record_sequence = [0, 100, 200, 300, 400, 0]

# record_sequence = [0, 100, 200, 300, 400, 0]
# Step size = 100; values loop back to 0
# Range spans 0 to 400 with step 100
variable_program_setting_popcorn = ContinuousVariable(
    value_ranges_steps=[(0, 400, 100)],
    current_value=0
) # in grams
program_setting_dict["popcorn"] = variable_program_setting_popcorn
```

Prompt 12: Update Appliance Model After Updating Variable

You are given:

- The original simulator implementation.
- The incorrect variable name, `variable_x`.
- The corrected variable definition.

Your Task: Update the `Simulator()` class so that all references to `variable_x` reflect its corrected definition.

Instructions:

(1) For `Simulator()`, edit only affected action methods. Keep unrelated parts of the simulator unchanged. Do not modify or omit the `reset()` method.

(2) Exclude code outside `Simulator()`, such as class definitions (`Appliance()`, `Variable()`), variables and `simulator_feature`.

Example Output:

```
# variable_power was changed from ContinuousVariable to
↪  DiscreteVariable. The valid value ranges change from float (e.g.
↪  100) to string (e.g. "100").
class Simulator(Appliance):
    def reset(self):
        ...

    def press_start_button(self):
        self.feature.update_progress("press_start_button")
        current_feature = self.feature.current_value[0]
        if current_feature == "speed_cook":
            self.assign_variable_to_next(self.variable_cooking_time)
            # updated line
            self.variable_power.set_current_value("100")
```

Prompt 13: Update Goal Value After Variable Definition Change

You are given a user instruction, an appliance model, a goal state object

- a user instruction.
- the implemented appliance model, i.e., a `Simulator()` object.
- A `goal_state = Simulator()` object specifying target variable values that achieves the instruction.
- The updated variable name, `variable_x`.
- A goal-setting guide for reference.

Your Task: Update the goal value of `variable_x` in the goal state to match the new definition.

Instructions:

(1) Ensure the new value assignment aligns with both the the updated definition `variable_x` and the user instruction.
(2) Do not rename `variable_x`. Do not modify any other variables in the goal state.
(3) Do not return any other content (e.g., comments, reasoning, variable definitions, or unrelated goal assignments).

Output Format: A single line of valid Python code that updates `goal_state.variable_x` to the correct value.

Example Output:

```
# updated timer to ContinuousVariable, previously was DiscreteVariable
goal_state.variable_microwave_timer.set_current_value(3) # minutes
```

Prompt 14: Check if Bounding Box Contains Control Panel Element

Task: Given an image labeled with a bounding box, determine whether the bounding box contains a control panel element.

Definition of Control Panel Element: Control panel elements include:

- Physical components: buttons, dials.
- Soft pads: labels printed directly on the control surface that respond to touch input. These labels might include printed symbols and icons, such as: ”+”, ”-”, ”start”, ”on/off”, and numeric digits.

Instructions:

(1) Review the region circled by the bounding box.
(2) If the bounding box contains any of the valid elements listed above, reply with `"Yes"`. Otherwise reply with `"No"`.
(3) In both cases, provide a reason by naming the object being circled by the red bounding box.

Output Format:

```
Yes
Reason: The red box surrounds the "+" symbol on the soft pad region.
```

Prompt 15: Map Bounding boxes to Control Panel Element Names

You are given:

- A list of control panel element names including buttons, dials, and soft-labeled pads.
- Three images:
  (1) Full view of the control panel.
  (2) Zoomed-in region with a red bounding box and several green bounding boxes.
  (3) Same zoomed-in region without bounding boxes.
- A `bounding_box_index` referring to the red box.

Your Task:

(1) Determine whether the red bounding box encloses a listed control element. Be lenient: if the red box contains any label, symbol, or visible control region, attempt to match. If multiple names match the red box, include them all.
   - For **dials**: Only bounding boxes covering the knob are valid. Ignore labels around the dial.
   - For **buttons**: Only bounding boxes that cover the physical, pressable area are valid. Boxes that only enclose external labels are invalid.
   - For **soft-labeled pads**: If the label itself is the interactive surface (i.e., no visible border or physical button), bounding boxes over the label region are valid.
(2) If (1) is true, check if the red box is a better match than any green box for the same element.

- It is okay for red box to partially enclose the object.
- If red box is clearer or more precise than all green boxes, accept it as the match.

Output Format:

- If both conditions are met, output the matched control element(s) in format below. Use exact names from the provided list.

```
<control_element_name> : <index>
<control_element_name> : <index>
...
```

- If no valid match is found, output `None`.

Example Output:

```
temperature dial : 1
power dial: 2
None
temperature dial: 3
power dial: 3
```

Prompt 16: Remove Duplicate Bounding Boxes for Control Panel Elements

You are given an `appliance_type`, which contains a `control_panel_element_name`. Control panel elements are components responsible for operating the appliance, such as buttons, dials and soft touch pads. You are given:

- A photo of the appliance to identify `control_panel_element_name`.
- A sequence of images showing bounding box options around potential regions for `control_panel_element_name`. Each box has a visible `index` at its bottom-right corner.

Your Task: Select **one** bounding box index that best matches the `control_panel_element_name`. If none of the bounding boxes is valid, return `response_index = -1`.

Selection Criteria:

- **Dial:** Choose the bounding box that covers the *knob*. Ignore boxes that only include labeling or surrounding text.
- **Button:**
  - If the label is printed directly on the button, a box selecting either the full button or label area is valid, even if the coverage is partial.
  - If the label is outside a physical button, select the bounding box around the physical (extruded) button, not just the label.
- **Soft Pad:** When the label text or icon *is* the button (i.e., not physically extruded), select the box that covers any part of that label or symbol.

Output Format: Return two variables in Python format:

```
response_index = 3
response_reason = "The bounding box covers the soft pad label text of
↪  the button."
```

If no bounding box fits the criteria:

```
response_index = -1
response_reason = "None of the boxes select the physical button or
↪  label. The target is a circular dial knob near the bottom left
↪  corner."
```

Prompt 17: Ground Actions

You are given a list of action names and a list of control panel element names. Your task is to ground each action to a control panel element name and a valid action type. Valid action types include `press`, `hold`, `turn_dial_clockwise`, `turn_dial_anti_clockwise`.

Output Format: Return a Python list of dictionaries. Each dictionary contains a grounded action, with the following keys:

(1) `"action"`: a string from the given action list (e.g., `"press_max_crisp_button"`).

(2) `"bbox_label"`: a list of strings from the given control element names.
- For standard actions, this is a single-element list (e.g., `["max_crisp_button"]`).
- For simultaneous actions (e.g., `hold_wash_button_and_rinse_button`), include both elements (e.g., `["wash_button", "rinse_button"]`).

(3) `"action_type"`: inferred from the action name string using the following rules:
- Contains `"hold"` ⇒ `"hold_button"`
- Contains `"press"` ⇒ `"press"`
- Contains `"turn_dial_clockwise"` ⇒ `"turn_dial_clockwise"`
- Contains `"turn_dial_anti_clockwise"` ⇒ `"turn_dial_anti_clockwise"`

Example Output:

```
[
    {
        "action": "press_max_crisp_button",
        "bbox_label": ["max_crisp_button"],
        "action_type": "press_button"
    },
    {
        "action": "press_and_hold_cancel_button_and_stop_button",
        "bbox_label": ["cancel_button", "stop_button"],
        "action_type": "press_and_hold_button"
    }
]
```

Prompt 18: Visual Feedback Parsing

You are given:
- A user command describing the task.
- The most recent action applied and the target variable being adjusted.
- The valid value range of the target variable.
- An image of the appliance control panel after the action.
- Relevant user manual text describing the display panel.

Your Task:
- Interpret the display image to infer the current appliance state, especially the value of the target variable.
- Use the user manual to explain display symbols if needed.

Output Format:

```
variable_description = "<Concise interpretation of the current state,
↪  focusing on the target variable.>"
```

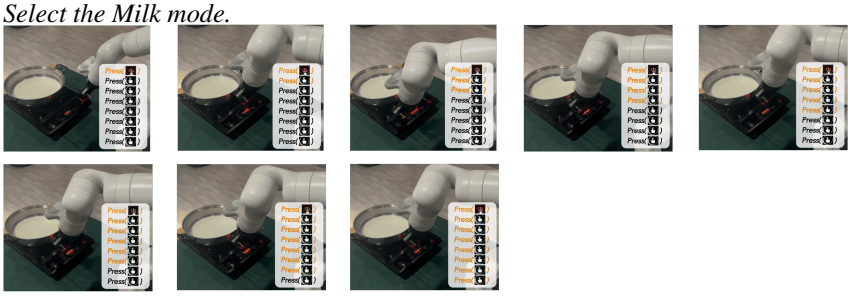
Example:

```
# Task: Set temperature to 98°C. Action: 'press_temp_clean_button'.
# Display shows a triangle under 85°C.

variable_description = "The triangle under '85°C' indicates the
↪  current selection. variable_temperature = 85."
```

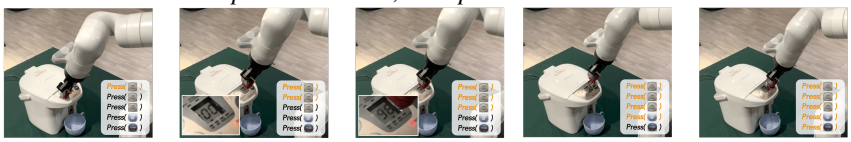
Select the HotPot mode and set power to 2000 W.

Select the Milk mode.



Select the HotPot mode and set power to 1600 W.

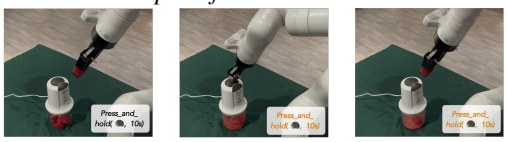
Set the insulation temperature to 98°, then pour the water.



Set the insulation temperature to 85°, then pour the water.

Set the insulation temperature to 65°, then pour the water.

Hold at slow speed for 10 seconds.



Hold at slow speed for 15 seconds.

Hold at turbo speed for 10 seconds.

Figure 18: Snapshots of our system performing various tasks on real appliances. Each row shows execution steps for one task.