

## 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  $\bar{A}$  should appear in some macro actions.
4. **Variable Coverage.** Every variable defined in the state space  $\bar{S}$  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:**  $\bar{S}_g$  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  $\bar{T}_g$  are mostly correct. Therefore, for each action in  $\bar{A}_n$ , 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  $\bar{A}_n$  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. 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



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

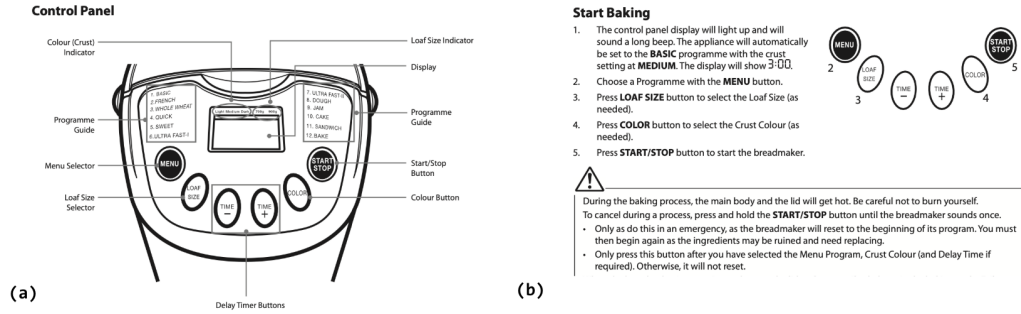


Figure 9: An example user manual for the Bread Maker. (a) Control panel. (b) Unstructured step-by-step procedure for a macro action.

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  $\rightarrow$  high  $\rightarrow$  low  $\rightarrow$  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 diagnose. 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 ensure fair comparison, we assign a fixed number of target variables per appliance type, subject 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

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 action, 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  $\bar{A}$ . The optimal number of actions is manually labeled by a human oracle. Intuitively, this metric evaluates the efficiency considering both execution and exploration.



Table 2: Hyperparameters of Used Models.

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

**Execution Step:** It indicates the average number of symbolic actions in  $\bar{A}$ , 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.  $\checkmark$  indicates the component is present in the corresponding method;  $\times$  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.

**Structured Model**  $\bar{\mathcal{M}}$  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  $\bar{\Phi}$  and the task policy  $\pi$ , instead of being chosen reactively by an LLM.

**Structured Reasoning via**  $\bar{\mathcal{T}}$  denotes whether action sequences for variable adjustment are computed using predefined transition functions  $\bar{\mathcal{T}}$ , rather than being generated by LLMs.

**Close-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: Ablation of key components in each method.  $\checkmark$  indicates the component is used.

Method	Grounded Action	Structured Model $\overline{\mathcal{M}}$	Structured Reasoning via $\overline{\mathcal{T}}$	Close-loop Update
LLM as policy w/ image	$\times$	$\times$	$\times$	$\checkmark$
LLM as policy w/ grounded actions	$\checkmark$	$\times$	$\times$	$\checkmark$
ApBot w/o structured model	$\checkmark$	$\times$	$\checkmark$	$\checkmark$
ApBot w/o structured reasoning	$\checkmark$	$\checkmark$	$\times$	$\checkmark$
ApBot w/o close-loop update	$\checkmark$	$\checkmark$	$\checkmark$	$\times$
ApBot	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$

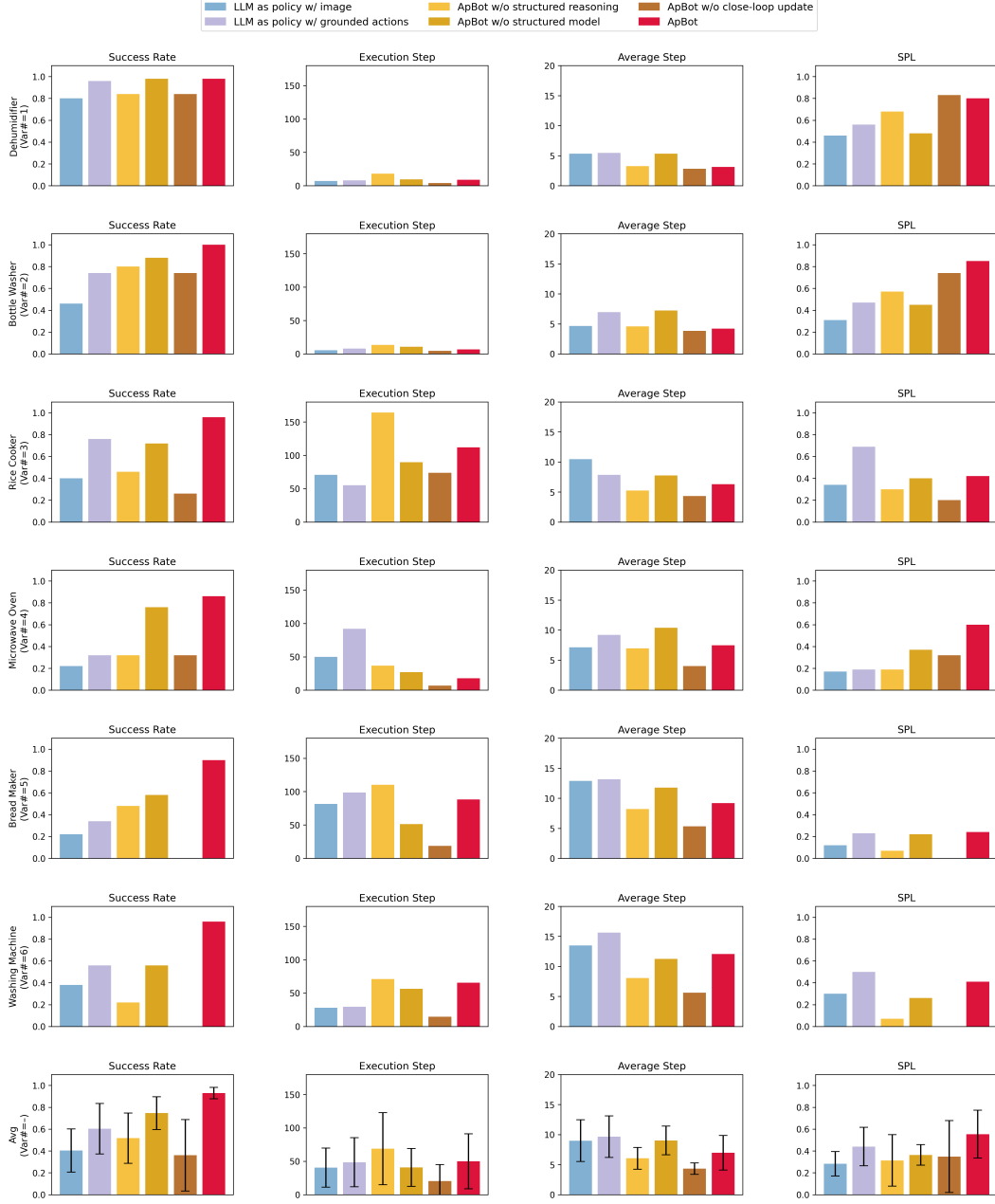


Figure 10: Performance of home appliance operation by appliance type, including average task success rate (SR), average number of execution steps (Average Steps), and SPL (Success weighted by Path Length).

## 661 F Details of Performance

662 Figure 10 shows the performance of ApBot on six appliance types. Each appliance type has a  
 663 different number of variables to adjust, from 1 to 6 (top to bottom). As the number of variables  
 664 increases, ApBot does not suffer a severe performance drop in terms of success rate (Figure 12). By  
 665 contrast, baseline methods like *LLM as policy w/ image* and *LLM as policy w/ grounded actions* drop  
 666 significantly on tasks with more variables. This shows that structured models, structured reasoning,  
 667 and closed-loop updates help in handling complex tasks. Another interesting observation is that  
 668 SPL suffers from an obvious drop when increasing the complexity of appliances. The reason is that  
 669 for complex appliances and tasks, there will always be more modeling errors, which require more

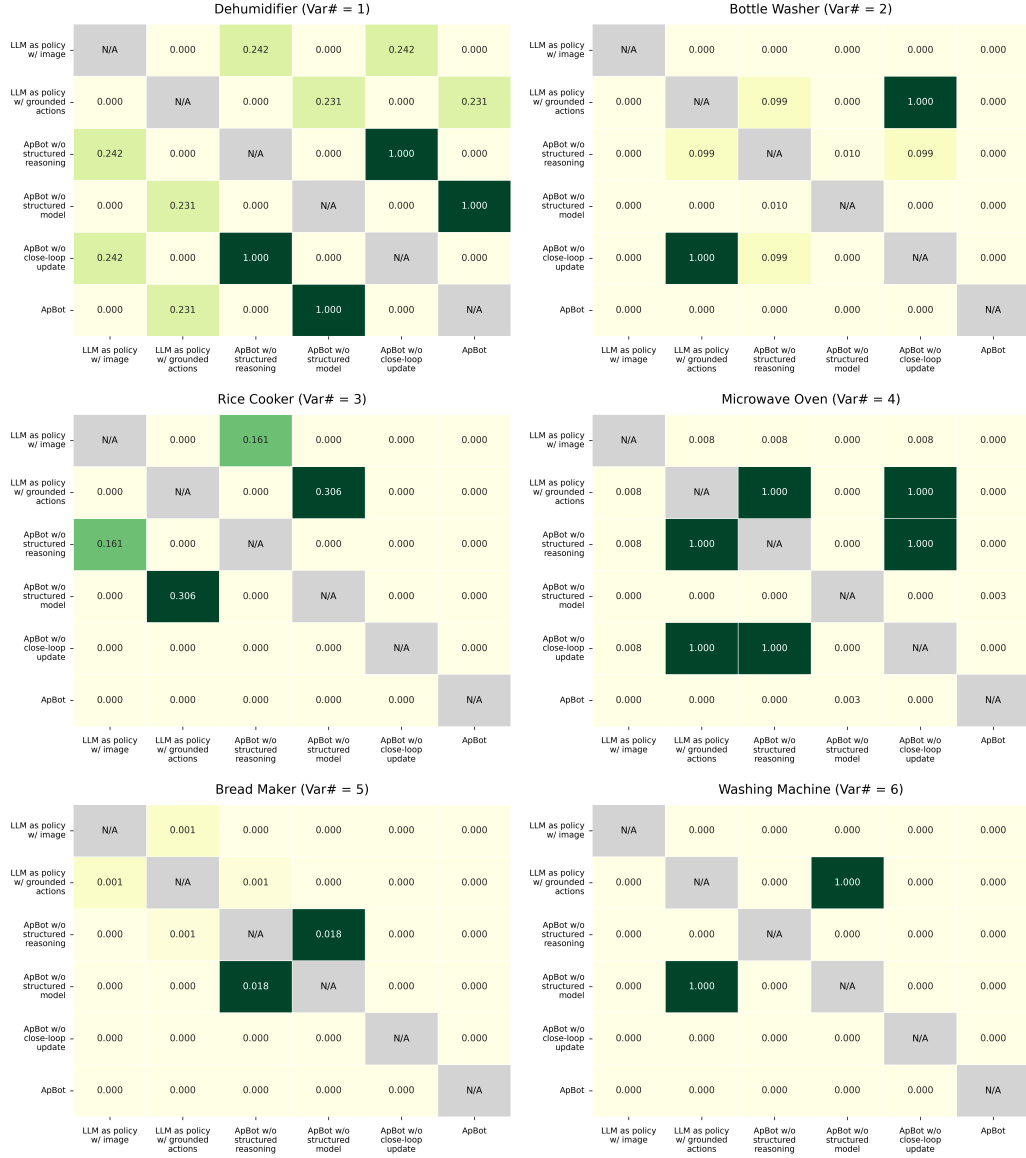


Figure 11: p-value matrix of all method pairs by  $\chi^2$ -test on different appliance types.

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

Figure 11 presents pairwise  $\chi^2$ -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 13 compares action grounding performance between ApBot and Molmo across six appliance types, evaluated by precision, recall, and F1 score. ApBot consistently outper-

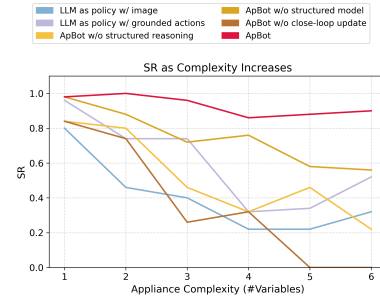


Figure 12: Average task success rate (SR) by increasing variable size conditioned on appliance type.

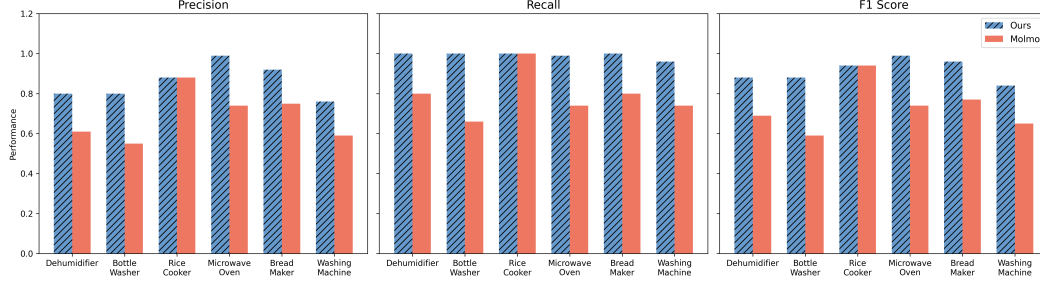


Figure 13: Comparison of action grounding performance between our method and Molmo on precision and recall across appliance types.

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

Method	Dehumidifier	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 structured reasoning	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 structured 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

forms Molmo on all 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:**

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

690



## G Details of Real World System

### G.1 Real World System Design

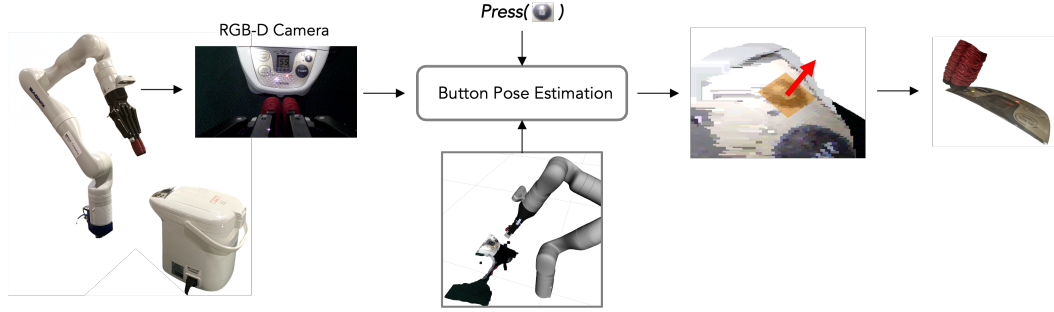


Figure 14: The real-world framework

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 14. 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.

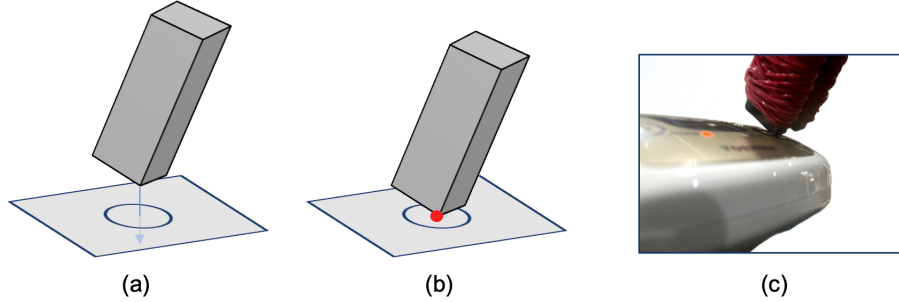


Figure 15: The pressing details

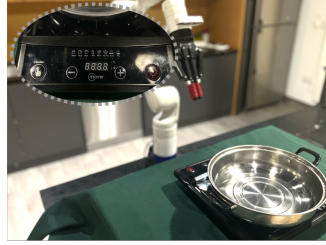
To reduce the contact area and improve precision, the robot aligns its gripper with the surface normal at a slight tilt. The pressing trajectory is generated in two stages: first, the end-effector moves to a position directly above the button; then, it advances 0.1 cm beyond the estimated button surface to ensure a firm press, as shown in Figure 15. 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

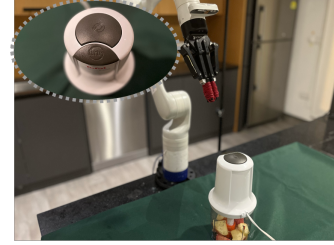
To evaluate the performance and generalization ability of our proposed system, we conducted a series of real-world experiments involving common household appliances. Specifically, the robot was tasked with operating three distinct devices: a blender, an induction cooker, and a water dispenser, as shown in Fig. 16. 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.



Water Dispenser



Induction Cooker



Blender

Figure 16: The real world setting

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

### 728 G.3 More Real World Execution Visualization

729 For more visualization, please see Fig. 17.

## 730 H Details of Failure Mode Analysis

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

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

737 **Failure due to Lack of Structured Model** This mainly include failures that occur due to (1) incorrect  
738 association between actions and effects due to lack of transition modeling; (2) repeated adjustment of the same  
739 variable due to lack of macro actions; (3) premature ending due to lack of state estimation; (4) incorrect goal  
740 state specification due to lack of structured goal states. This failure mode is mainly applicable to LLM as policy  
741 w/ image; LLM as policy w/ grounded actions; ApBot w/o structured model.

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

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

## 749 I Prompts

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

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

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

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

### Prompt 1: LLM as policy w/ image Action Proposal

You are given:

- Two images:
  - (1) A photo of the appliance contrl 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.

769

#### 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.
- 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
```

770

### 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.

771

- 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)"
```

772

### 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:**

773

```
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
]
```

774

## 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
```

775

## 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:

776



- (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.

```

779

## 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))

```

780

### 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 `meta_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"}]
```

782

## 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:

783

- 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_lockbutt
        ↪ 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"`  $\Rightarrow$  True; `variable_power = "0n"` vs. `variable_on_off = "0n"`  $\Rightarrow$  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"  $\Rightarrow$  True; "100cm" vs. "100"  $\Rightarrow$  True; "1 cup" vs. "1 serving"  $\Rightarrow$  True
- (3) Ensure the match is the closest in the value range. *E.g.* `program="wash"` vs. `program="wash, dry"`, both values exist in value range  $\Rightarrow$  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
```

789

## 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:

790

```

# 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, ...]

```

791

## 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:

792



```

# 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

```

793

## 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")

```

794

## 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.

795

**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
```

796

### 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.
```

797

### 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.

798

- 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
```

799

## 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."
```

800

## 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"`  $\Rightarrow$  `"hold_button"`
  - Contains `"press"`  $\Rightarrow$  `"press"`
  - Contains `"turn_dial_clockwise"`  $\Rightarrow$  `"turn_dial_clockwise"`
  - Contains `"turn_dial_anti_clockwise"`  $\Rightarrow$  `"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"
  }
]
```

801

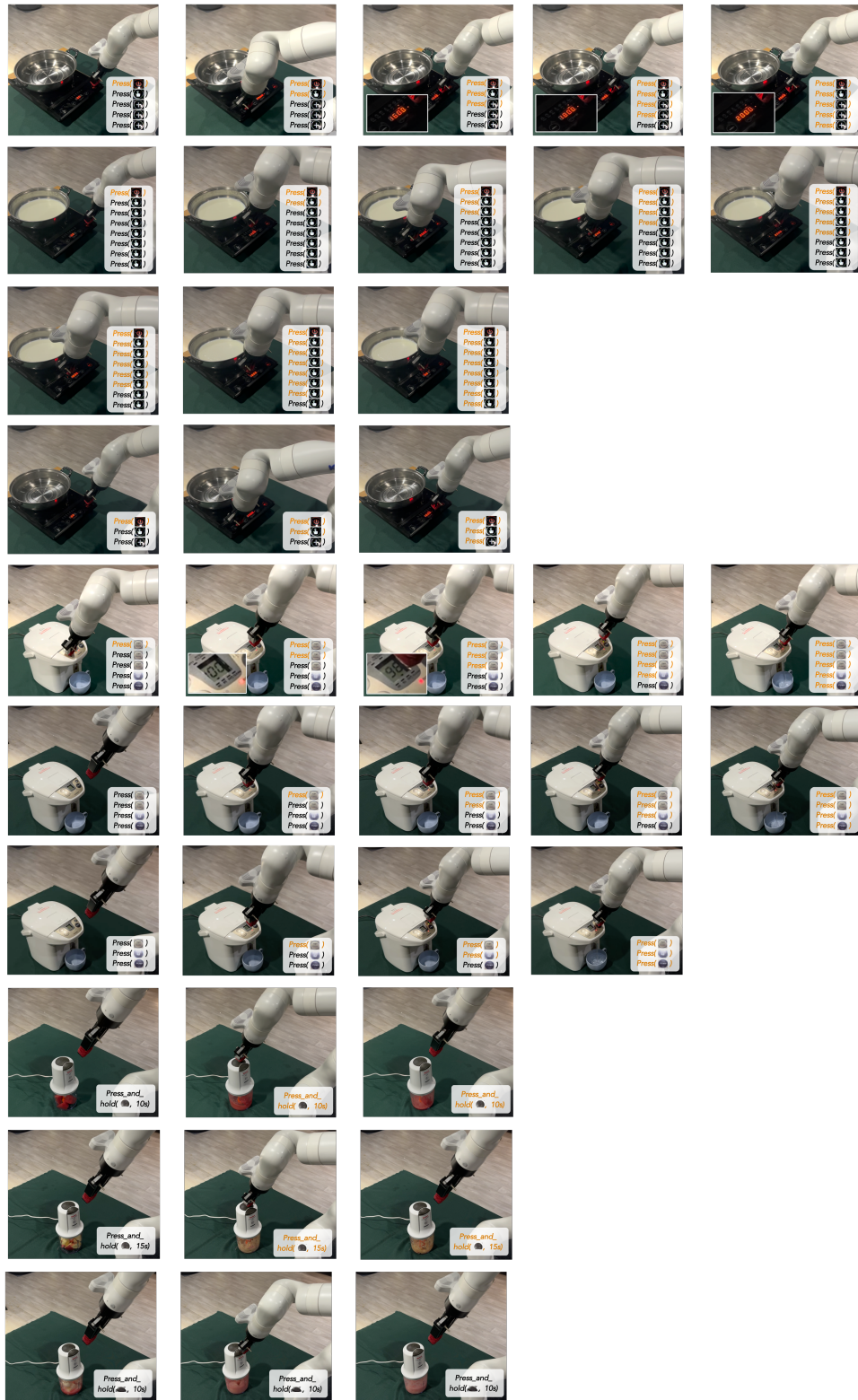


Figure 17: Snapshots of our system performing all the tasks.