Multimodal Pretrained Models for Verifiable Sequential Decision-Making: Planning, Grounding, and Perception

Yunhao Yang Cyrus Neary Ufuk Topcu
The University of Texas at Austin
Austin, TX, 78712
{yunhaoyang234, cneary, utopcu}@utexas.edu

Abstract

Recently developed multimodal pretrained models can encode rich world knowledge expressed in multiple modalities, such as text and images. However, the outputs of these models cannot be integrated into algorithms to solve sequential decision-making tasks. We develop an algorithm that utilizes the knowledge from the pretrained models to construct and verify controllers for sequential decision-making tasks and to ground these controllers to task environments through visual observations. In particular, the algorithm constructs an *automaton-based controller* that encodes the task-relevant knowledge extracted from the pretrained model. It then verifies whether the knowledge encoded in the controller is consistent with other independently available knowledge, which may include abstract information on the environment or user-provided specifications. If this verification step discovers any inconsistency, the algorithm automatically refines the controller to resolve the inconsistency. Next, the algorithm leverages the vision and language capabilities of pretrained models to ground the controller to the task environment. We demonstrate the algorithm's ability to construct, verify, and ground automaton-based controllers through a suite of real-world tasks.

1 Introduction

The rapidly emerging capabilities of multimodal pretrained models, also known as foundation models, in question answering, code synthesis, and image generation offer new opportunities for autonomous systems. However, a gap exists between the text-and-image-based outputs of these models and algorithms for solving sequential decision-making tasks. Additional methods are required to integrate the outputs of these foundation models into autonomous systems that can perceive and react to an environment in order to fulfill a task. Additionally, it is hard, if not impossible, to formally verify whether autonomous systems implementing such foundation models satisfy user-provided specifications.

Towards closing the gap between foundation models and sequential decision-making algorithms, we develop a pipeline that integrates the outputs of foundation models into downstream design steps, e.g., control policy synthesis or reinforcement learning, and provides a systematic way to ground the knowledge from such models. Specifically, we first develop an algorithm named `LLM2Automata` to construct *automaton-based controllers* representing the knowledge from the foundation models. It queries the foundation model to obtain text-based task knowledge, parses the text to extract actions, and defines a set of rules (grammar) to transform these actions into a *finite state automaton* (FSA).

The automaton-based representations can be formally verified against knowledge from other independently available sources. This verification step ensures consistency between the knowledge encoded

Foundation Models for Decision Making Workshop at NeurIPS 2023.

in the foundation model and the knowledge from other independent sources. If the verification step fails, we present a procedure to automatically refine the controller to resolve any potential inconsistencies. The independently available knowledge may be explicitly given by the user in an automaton-based form that is compatible with the verification step, or it may be provided in textual form (e.g., user manuals, online information, or natural-language descriptions). In the latter case, we develop an algorithm `Text2Model` that automatically builds such automaton-based representations encoding the text-based independently available knowledge.

We then propose a method `Automata2Env` that leverages the multimodal capabilities of the foundation models, i.e., simultaneous vision and language understanding, to ground these controllers through visual perception. `Automata2Env` collects visual observations and uses the vision and language capabilities of the employed foundation model to evaluate the truth values of conditions from the controller. The controller then uses these truth values to select its next action.

We demonstrate the algorithms' capabilities on sequential decision-making tasks through a variety of case studies (e.g., cross the road). The examples show the algorithms' ability to construct verifiable knowledge representations and to ground these representations in real-world environments through visual perceptions.

2 Related Work

Formal Representations of Textual Knowledge. Many works have developed methods to construct symbolic representations of task knowledge from natural language descriptions. Several works construct knowledge graphs from textual descriptions of given tasks [West et al., 2022, Rezaei and Reformat, 2022, He et al., 2022], or analyze causalities between the textual step descriptions and build causal graphs [Lu et al., 2022]. However, the graphs resulting from these works are not directly useful in algorithms for sequential decision-making, nor are they formally verifiable. Another work builds automaton-based representations of task-relevant knowledge from text-based descriptions of tasks Yang et al. [2022]. In contrast with Yang et al. [2022], we not only generate automaton-based representations but also ground the generated representations to the task environment through image-based perceptions.

Foundation Models in Sequential Decision-Making. A work [Lu et al., 2023] generates static high-level plans and matches them to the closest admissible action. Some other works [Huang et al., 2022, Lin et al., 2023, Liu et al., 2023a, Singh et al., 2023] generate zero-shot plans for sequential decision-making tasks from querying generative language models. These works require a set of pre-defined actions, which limit their generalization capability. Another work [Vemprala et al., 2023, Ichter et al., 2022] uses multimodal foundation models to generate executable code or API for robots based on visual perceptions from the environment. These works lack a procedure to ensure the correctness or safety of their generated plans or executable actions. In contrast, the automaton-based representation we constructed enables others to formally verify the plans against some mission or safety specifications. Additionally, we provide automatic procedures to refine the plans to ensure the specifications will eventually be satisfied.

3 Preliminaries

Multimodal Pretrained Models. Multimodal pretrained models, also known as foundation models, are capable of processing, understanding, and generating data across multiple formats, such as images, text, and audio. These models are pretrained on large training datasets, and they have demonstrated strong empirical performance across a variety of tasks, such as question-answering and next-word prediction, even without further task-specific fine-tuning Brown et al. [2020].

The Generative Pretrained Transformer (GPT) series of models [OpenAI, 2023, Brown et al., 2020] consists of the most well-known multimodal pretrained models that can generate natural language or other data formats. In addition to GPT, pretrained models such as PaLM [Chowdhery et al., 2022], BLOOM [Scao et al., 2022], Codex [Chen et al., 2021], and Megatron [Smith et al., 2022] also have the capability of generating outputs in natural language or other formats. Language generation is the core capability of these models, which we will use in the rest of the paper. Hence we explicitly denote them as *Large-scale generative language models* (GLMs).

Vision-language models such as CLIP [Radford et al., 2021] and the Segment Anything Model [Kirillov et al., 2023] are another type of multimodal pretrained model. CLIP takes an image and a set of texts as inputs and measures the image-text consistency. The Segment Anything Model can be used for object detections, which take an image and a set of words that describe objects and classify whether the objects appear in the image.

Finite State Automaton. A *finite state automaton* (FSA) is a tuple $\mathcal{A} = \langle \Sigma, \Gamma, Q, q_0, \delta, \omega \rangle$ where Σ is the input alphabet (the set of input symbols), Γ is the output alphabet (the set of output symbols), $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \times Q \to \{0, 1\}$ is the transition function, and $\omega : Q \times \Sigma \times Q \to \Gamma$ is the output function.

We use P to denote the set of atomic propositions, which we use to define the input alphabet, $\Sigma \coloneqq 2^P$. In words, any given input symbol $\sigma \in \Sigma$ consists of a set of atomic propositions from P that currently evaluate to True. A *propositional logic formula* is based on one or more atomic propositions in P. A transition from q_i to q_j exists if $\delta(q_i, \varphi, q_j) = 1$, the current state is q_i, and the propositional logic formula φ is true. Note that we define the FSA transitions to possibly be non-deterministic—mulitple transitions are possible under the same input symbol, from a given FSA state.

Controllers and Models In this work, we refer to the automaton-based representation of task knowledge as a *controller*: a system component responsible for making decisions and taking actions based on the system's state. A controller is represented as mapping the system's current state to an action, which can be interpreted as a control input or a setpoint. Mathematically, we use an FSA $\langle \Sigma, \Gamma, Q, q_0, \delta, \omega \rangle$ to represent the controller, whose input alphabet Σ indicates all possible observations of the environment and output alphabet Γ indicates all possible actions. We additionally allow for a "no operation" action $\epsilon \in \Gamma$.

The controller's goal is to adjust the control input so that the system's state evolves in a way that satisfies externally provided requirements or properties. These requirements or properties are often specified using formal languages, such as *linear temporal logic* (LTL) [Biggar and Zamani, 2020].

A *model* is a transition system that may represent either the dynamics of the task environment, or knowledge from other independent sources. We formally define a *model* as $\mathcal{M} \coloneqq \langle \Sigma_{\mathcal{M}}, \Gamma_{\mathcal{M}}, Q_{\mathcal{M}}, \delta_{\mathcal{M}}, \omega_{\mathcal{M}} \rangle$, which consists of input alphabet $\Sigma_{\mathcal{M}} \coloneqq 2^{P_{\mathcal{M}}}$ is a set of input symbols, where $P_{\mathcal{M}}$ is defined as the actions. $Q_{\mathcal{M}}$ is a finite set of states, $\delta_{\mathcal{M}} : Q_{\mathcal{M}} \times \Sigma_{\mathcal{M}} \times Q_{\mathcal{M}} \to \{0, 1\}$ is a non-deterministic transition function, and $\omega_{\mathcal{M}} : Q_{\mathcal{M}} \to \Gamma_{\mathcal{M}}$ is a labeling function, where $\Gamma_{\mathcal{M}} = 2^{\overline{P}}$ and $\overline{P}$ is a set of atomic propositions representing conditions of the environment.

4 Task Controller Construction, Refinement, and Grounding

We develop an algorithm, `LLM2Automata`, that takes a brief task description in textual form from the task designer and returns an FSA representing the task controller that can be verified against specifications given by the task designer.

We then design a verification-refinement procedure for the controller constructed through `LLM2Automata`. We first obtain a model representing the dynamics of the task environment, or other side information from external knowledge sources other than the GLM. This model could be created by a human designer, or it can be automatically constructed from the textual information contained in external knowledge sources such as websites, operation manuals, blog posts, or books. We present the algorithm for such automatic construction in Section B.1 below. Next, we verify whether the controller, when implemented against the model, satisfies the user-defined specifications. If the verification step fails, we use the verification outcomes, e.g., counter-examples, to refine the controller until the controller passes the verification step.

4.1 Controller Construction

The algorithm `LLM2Automata` takes a brief text description of a task and constructs an FSA to represent the controller of the given task. Specifically, the algorithm sends the text description as the input prompt (in blue) to a GLM and obtains the GLM's

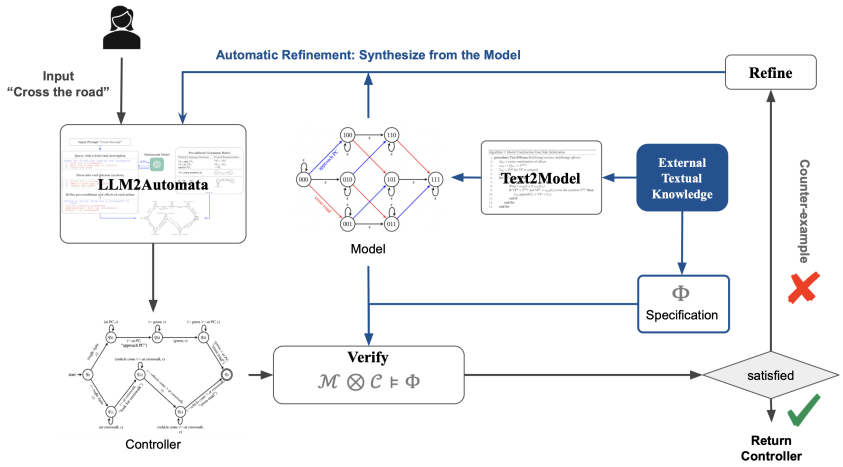


Figure 1: Demonstration of constructing, verifying, and iteratively refining the FSA-based controller of a given task.

response (in red), which is a list of steps for achieving the task in textual form:

```
Steps for task description
step_number_1. step description
step_number_2. step description
...
```

The algorithm uses the semantic parsing method introduced in GLM2FSA [Yang et al., 2022] to parse each step description into *verb phrases (VP)* and connective keywords. A list of pre-defined keywords is provided in Table 1. A verb phrase consists of a verb and its noun dependencies. Each step corresponds to a state in the FSA. Meanwhile, each verb phrase VP in the step description represents an *action*, and the algorithm queries the GLM to extract the precondition and effect of this action in the form of *Planning Domain Definition Language* (PDDL):

```
Define an action "action name" in PDDL
 Action: action name
 Precondition: a set of propositions
 Effect: a set of propositions
```

We use the extracted verb phrase VP_i to define the action name, and we use VP_i^C and VP_i^E to denote the precondition and effect of the action, respectively. Then, the algorithm follows the rules illustrated in Table 1 to transform natural language into propositions or automaton transitions. Each step description is translated into a state in the FSA and a set of outgoing transitions from this state.

We note that in contrast to the GLM2FSA algorithm presented in Yang et al. [2022], we query the GLM for the preconditions and effects of each action and encode them into the constructed controller. These preconditions and effects are descriptions of the task environment prior to and after taking some actions. This explicit representation of the actions' preconditions and effects is required for the subsequent methodology we propose to ground the constructed automaton-based controllers to their task environments via image-based observations, described in Section 4.3.

4.2 Verification and Refinement

After constructing the controller, users can verify the controller against other independently available knowledge, which may include abstract information on the environment or user-provided specifications. This verification provides a formal way of checking the "correctness" of the controller—whether the knowledge encoded in the controller is consistent with the external knowledge. By doing so, we use a *model* to encode the external knowledge.

A model is a transition system that encodes the dynamics of the task environment or the task-relevant knowledge from external knowledge sources. If we obtain an automaton-based model, we can use

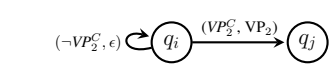
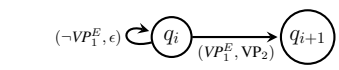
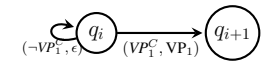
Natural Language Grammar	Formal Representation	Example
VP_1 **and** VP_2	$\text{VP}_1 \wedge \text{VP}_2$	[green light] [and] [no car]
no/not VP_1	$\neg\ \text{VP}_1$	[no] [car]
if VP_1, VP_2. VP_2 **if** VP_1		[if] [green light], [cross]
wait VP_1 VP_2 VP_2 **after** VP_1		[wait] [green light] [cross]
VP_1		[cross road]

Table 1: Rules to convert natural language grammar to formal representations. The keywords that define the grammar are in bold. The complete grammar table is in the Appendix.

the model directly for the verification steps. However, in many cases, external knowledge is only available in textual form, such as websites and operation manuals. Again, such textual information is not formally verifiable. To address this problem, we develop an algorithm named TEXT2MODEL (as presented in Algorithm 1 in the Appendix) that automatically constructs a model that encodes the textual information from external sources.

Once we have the controller and the model, we use the model to formally verify whether the controller satisfies user-provided specifications and design procedures to refine the controller if the verification step fails. We additionally present an automatic refinement method that makes more explicit use of the model.

Automated Verification. In the verification procedure, we build a product automaton $\mathfrak{P} = \mathcal{M} \otimes \mathcal{C}$ describing the interactions of the controller $\mathcal{C}$ with the model $\mathcal{M}$. We define the product automaton in the Appendix.

Then, we obtain a *specification* Φ expressed in *linear temporal logic* from the task designer or whoever wants to verify the controller. We run a model checker (e.g., NuSMV [Cimatti et al., 2002]) to verify if the product automaton satisfies the specification,

$$\mathcal{M} \otimes \mathcal{C} \models \Phi. \tag{1}$$

We verify the product automaton against the specification for all its possible initial states. If the verification fails, the model checker returns a counter-example, which is a sequence of states of the product automaton $(p_1, q_1), (p_2, q_2), ...$ where $p_i \in Q_{\mathcal{M}}, q_i \in Q$.

If the verification fails, we iteratively refine the controller until the specification holds for all the initial states. The refinement procedure starts from the counter-example returned by the model checker and eventually produces a refined controller that will never violate the specification.

Automatic Refinement to Resolve Inconsistencies. Once we obtain a counter-example, we use the model to synthesize a new FSA that represents a sub-controller. This automatic refinement is designed to ensure the liveness of the controller, i.e., some goals will be achieved eventually.

Specifically, we run the model checker on the model against the negation of the specification and set the initial state to the first state in the counter-example. Mathematically, the model checker verifies

$$\mathcal{M} \models \neg\Phi. \tag{2}$$

The model checker returns a counter-example $\overline{Q} = \{p_1, ..., p_n\}$ to the negated specifications, which may be interpreted as an example that satisfies the specification. This automatic refinement is feasible only if the length of $\overline{Q}$ is finite. This counter-example is a sequence of states p_i from the model. Such state sequence can be converted into a trajectory $T = (p_1, a_1), ..., (p_n, a_n)$, where $p_i \in 2^{\overline{P}}$ are the state labels and $a_i \in \Sigma_{\mathcal{M}}$ are the input symbols. We construct an FSA that has an identical number of states as the number of steps in $\overline{Q}$. The states for the new FSA are $\overline{q_i}$ for $i \in [0, ..., |\overline{Q}|]$. The transition function of the FSA takes $p_i \in 2^{\overline{P}}$ as input symbols and adds a transition from $\overline{q}_{i-1}$ to $\overline{q}_i$.

The label function takes the states and input symbols $(\overline{q}_{i-1}, p_i, \overline{q}_i)$ and outputs $a_i \in \Sigma_{\mathcal{M}}$. We have now constructed an FSA representing the desired sub-controller.

Next, we merge the new sub-controller into the original controller. We begin by creating a "no operation" transition whose input symbols are the labels of the first state in the negated counter-example. Then, we modify the input of the self-transition in the controller's initial state to the conjunction of the negations of the input symbols of its outgoing transitions. Finally, if the last state of the sub-controller does not have any outgoing transitions, we add a "no operation" transition from it to any of the *sink* states in the controller. Note that a sink state is a state that can only transit to itself. Now, the controller and the sub-controller together form a refined controller.

The details of the refinement procedure are in Algorithm 2 in the Appendix. We iteratively repeat this verification and refinement procedure until the specification is satisfied for all the initial states.

4.3 Grounding and Perception

We develop a method named `Automata2Env` to ground the controller to the real-world task environment. `Automata2Env` takes visual observations from the task environment and uses a vision-language model to determine the truth values of the atomic propositions that are relevant to the conditions specified in the controller.

To operate in the task environment, an agent starts from the initial state of the controller. The agent collects all the propositions P from the controller and gets an image observation from the task environment. It then feeds the image and all the propositions into a vision-language model. `Automata2Env` requires vision-language models that can output normalized scores indicating how each proposition matches the image (e.g., CLIP [Radford et al., 2021]). We refer to such scores as *confidence scores*, which are commonly provided as outputs of vision-language models. A higher score means the vision-language model is more confident that the context of the proposition is within the content of the image. The algorithm incorporates these confidence scores to evaluate the propositions.

Specifically, the algorithm takes an atomic proposition in textual form, a vision-language model that can return confidence scores, and numerical thresholds t as inputs. Recall that an input symbol is a set of atomic propositions. Then, the algorithm evaluates an atomic proposition and assigns one of the values true or false. Let the confidence score of an atomic proposition AP be c, and we assign $AP = true$ if $c \geq t$ and $AP = false$ otherwise.

After evaluating the set of atomic propositions, the agent chooses one transition whose input symbol (which itself is a logical formula over the atomic propositions) evaluates to true and takes corresponding actions.

5 Empirical Demonstration

We illustrate the `LLM2Automata` algorithm and the grounding method `Automata2Env` with a proof-of-concept example. We present the outputs of the algorithm step by step to show how it constructs, verifies, and refines the task controller. We also collect image observations from real-world environments to demonstrate the controller's behaviors in realistic deployment settings. We use the current state-of-the-art language model GPT-4 [OpenAI, 2023] and *Grounded-Segment-Anything* (Grounded-SAM) [Kirillov et al., 2023, Liu et al., 2023b] to produce the results in this section.

5.1 Cross Road Example

We start the demonstration on a daily life task: cross the road. In this example, we construct a controller that can handle the crossing-road task both at the traffic light and without a traffic light.

Controller Construction. First, we query GPT-4 for the steps of crossing the road at the traffic light and obtain a list of steps in textual form:

```
Steps for "cross the road at a traffic light"
 1. Approach the pedestrian crossing.
 2. Wait for the traffic light to turn green.
 3. Cross the road.
```

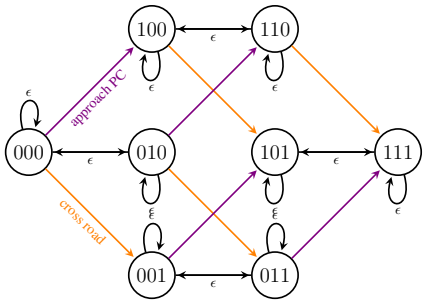


Figure 2: Transition system that represents the environment of the “crossing road at the traffic light" task. Transitions in violet and orange represent the transitions with actions “approach pedestrian crossing" and “cross-road," respectively. The label on the state indicates the value of propositions (“at PC," “green," “at other side"). For instance, 000 indicates $\neg$*at PC* $\wedge$ $\neg$*green* $\wedge$ $\neg$*at other side* and 010 indicates $\neg$*at PC* $\wedge$ *green* $\wedge$ $\neg$*at other side*.

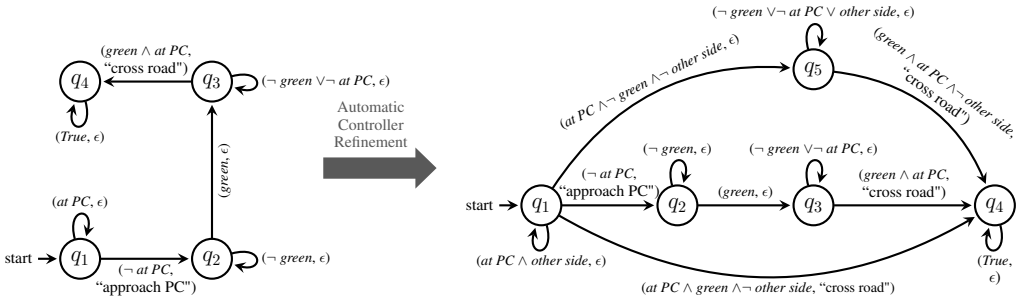


Figure 3: Controllers for the “crossing the road at a traffic light" task before (left) and after (right) the automated refinement procedure.

We parse the steps to extract the keywords and verb phrases and query GPT-4 again to define the verb phrases in PDDL and extract the preconditions and effects:

```
Define an action "approach pedestrian crossing" in PDDL
Action: Approach_pedestrian_crossing
Precondition: (not (at_pedestrian_crossing))
Effect: (at_pedestrian_crossing)

Define an action "Traffic light turn green" in PDDL
Action: Traffic_light_turn_green
Precondition: (not (traffic_light_is_green))
Effect: (traffic_light_is_green)

Define an action "cross road" in PDDL
Action: Cross_road
Precondition: (traffic_light_is_green) (at_pedestrian_crossing)
Effect: (at other_side)
```

We follow the grammar in Table 1 to transform each step into a state and its outgoing transitions. We get an FSA that represents the controller by connecting all the states with the transitions, which we present in Figure 3 (left).

Verification and Automatic Refinement. We use a user-provided model (as presented in Figure 2) to verify whether the controller satisfies a specification

$$\phi = (\Diamond \text{ green} \rightarrow \Diamond \text{ other side of road }) \wedge \Box(\neg \text{ cross road } \vee \text{ green }).$$

We present the details of how to obtain the model in the Appendix.

Intuitively, we want the agent to eventually reach the other side of the road but never cross the road when the traffic light is not green. We perform the verification step and get a counter-example. Hence we need to refine the controller.

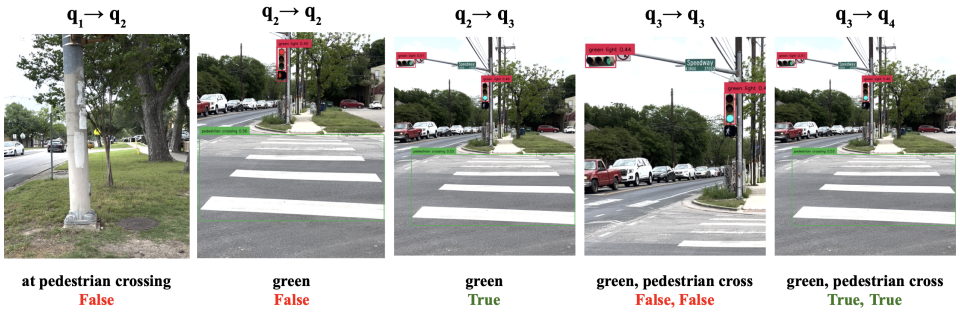


Figure 4: A sequence of observations from the real-world environment, where red and green boxes with confidence scores above are the object detection results from the Grounded-SAM.

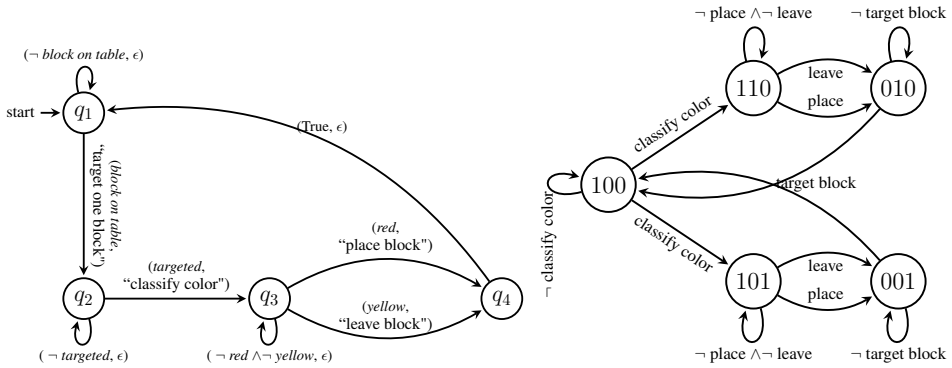


Figure 5: The left figure is the controller for "removing all the red blocks on the table," and the right figure is the model that encodes how the environment changes by the actions. The label on the model's state indicates the value of propositions ("block targeted," "red," "yellow"), where 1 indicates True and 0 indicates False.

In the first iteration of refinement, we get a counter-example whose trajectory starts from [at Pedestrian Crossing ∧¬ green ∧¬ the other side of the road], which is the "100" state in Figure 2. This counter-example means that if the agent is already at the pedestrian crossing when the controller is instantiated, it will get trapped in the initial state. This is because the precondition for the outgoing transition is "not at the pedestrian crossing". We follow Algorithm 2 to refine the controller before verifying it against the model again.

In the second iteration, we get another counter-example whose trajectory starts from [at Pedestrian Crossing ∧ green ∧¬ the other side of the road], which is the "110" state. We apply the refinement algorithm again and obtain the final controller that satisfies all the specifications. We present the final controller as the bottom automaton in Figure 3. The top and bottom branches of the final controller are synthesized during the first and second iterations, respectively.

Grounding to Task Environment. We use the Grounded-SAM to evaluate the input symbols and implement the control logic in the real-world task environment. The VLM takes an image and a set of propositions in textual form as inputs and classifies which propositions match the image. A proposition matches an image if the object or scenario described by the proposition appears in the image and is detected by the VLM with a confidence score above 0.45. We provide a visual demonstration in Figure 4.

5.2 Robot Arm Manipulation

We now use `LLM2Automata` to construct a controller for the task "use a robot arm to remove all the red blocks off the table". We show how we use the Grounded-SAM to perceive the operating environment and make decisions accordingly.

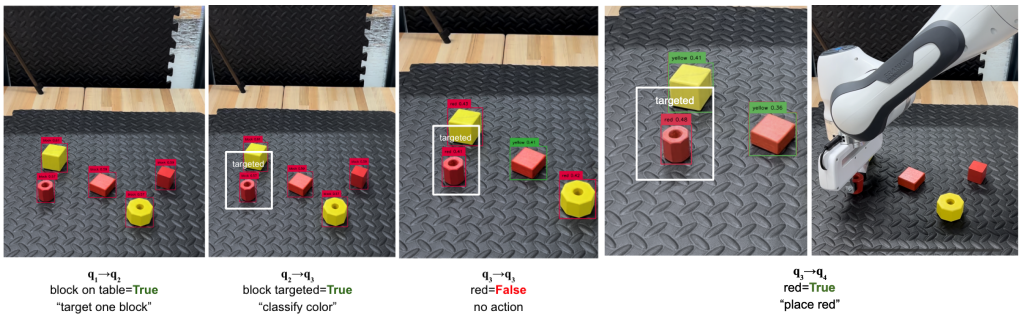


Figure 6: The figures show the object detection results from the VLM in red and green boxes, the proposition evaluation results, and the state transitions and actions that are taken under the evaluated propositions.

Controller Construction and Verification. In this example, we assume the user has some prior knowledge of the task, such as some basic knowledge of the environment and the admissible actions of the robot arm. The user thus queries GPT-4 with the following prompt:

```
Task: place all the red blocks off the table.
Environment: there are unknown numbers of red blocks and yellow blocks on
    the table initially. Someone may randomly add a red block or yellow
    block to the table.
Steps for achieving the task:
 1. Target one block on the table.
 2. Classify the color of the targeted block.
 3. If the block is red, place it from the table to an off-table location
    (B). If the block is yellow, leave it on the table.
 4. Go to step 1.
```

The complete queries and responses are presented in the Appendix. We construct the controller and present it in Figure 5.

Next, we verify whether the controller, when implemented in a model (right figure in Figure 5), satisfies the provided specification $\Phi = \neg place \wedge yellow$. As an added task specification, we want to guarantee the robot arm never accidentally places a yellow block outside the table. We also present the details of obtaining the model in the Appendix. The verification result shows that the controller satisfies Φ. Hence no refinement is required.

Grounding and Perception. We again use Grounded-SAM as the perception model to ground the controller from Figure 5 to the operating environment. We set the threshold t to 0.45. Figure 6 shows a full iteration of the controller ($q_1 \to q_2 \to q_3 \to q_4$). The robot arm grabs a block and places it outside the table only if the Grounded-SAM returns a confidence score above 0.45 on a red block.

We present more examples in the Appendix.

6 Conclusion

We provide a proof-of-concept for the automatic construction of an automaton-based task controller of task knowledge from GLMs and the grounding of the controller to physical task environments. We propose an algorithm that uses foundation models for sequential decision-making in the aspects of synthesis, verification, grounding, and perception. The algorithm synthesizes automaton-based controllers from the text-based descriptions of task-relevant knowledge that are obtained from a GLM. Such automaton-based controllers can be verified against user-provided specifications over models representing the task environments or task knowledge from other independent sources. The algorithm provides a method to iteratively refine the controller until all the specifications are satisfied. Additionally, we develop a grounding method `Automata2Env` that grounds the automaton-based controllers to physical environments, uses vision-language models to interpret visual perceptions, and implements control logic based on the perceptions. Experimental results demonstrate the capabilities of `LLM2Automata` and `Automata2Env` on synthesis, verification, grounding, and perception.

References

Hamza Ali. How to cross the road safely, Mar 2023. URL https://www.hseblog.com/cross-road-safely/.

Oliver Biggar and Mohammad Zamani. A framework for formal verification of behavior trees with linear temporal logic. *IEEE Robotics and Automation Letters*, 5(2):2341–2348, 2020. doi: 10.1109/LRA.2020.2970634. URL https://doi.org/10.1109/LRA.2020.2970634.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, and et al. Evaluating large language models trained on code. *ArXiv preprint arXiv:2107.03374*, 2021.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, and et al. Palm: Scaling language modeling with pathways. *ArXiv preprint arXiv:2204.02311*, 2022.

Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002. doi: 10.1007/3-540-45657-0_29. URL https://doi.org/10.1007/3-540-45657-0_29.

Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019.

Mutian He, Tianqing Fang, Weiqi Wang, and Yangqiu Song. Acquiring and modelling abstract commonsense knowledge via conceptualization. *arXiv preprint arXiv:2206.01532*, 2022.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 9118–9147, Baltimore, Maryland, USA, 2022. PMLR.

Brian Ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes, Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. Do as I can, not as I say: Grounding language in robotic affordances. In *Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pages 287–318, Auckland, New Zealand, 2022. PLMR.

Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment anything. *arXiv preprint arXiv:2304.02643*, 2023.

Kevin Lin, Christopher Agia, Toki Migimatsu, Marco Pavone, and Jeannette Bohg. Text2motion: From natural language instructions to feasible plans. *ArXiv preprint arXiv:2303.12153*, 2023.

B. Liu, Yuqian Jiang, Xiaohan Zhang, Qian Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+p: Empowering large language models with optimal planning proficiency. *ArXiv preprint arXiv:2304.11477*, 2023a.

Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Chunyuan Li, Jianwei Yang, Hang Su, Jun Zhu, et al. Grounding DINO: marrying DINO with grounded pre-training for open-set object detection. *arXiv preprint arXiv:2303.05499*, 2023b.

Yujie Lu, Weixi Feng, Wanrong Zhu, Wenda Xu, Xin Eric Wang, Miguel Eckstein, and William Yang Wang. Neuro-symbolic procedural planning with commonsense prompting. *arXiv preprint arXiv:2206.02928*, 2022.

Yujie Lu, Weixi Feng, Wanrong Zhu, Wenda Xu, Xin Eric Wang, Miguel Eckstein, and William Yang Wang. Neuro-symbolic procedural planning with commonsense prompting. In *International Conference on Learning Representations*, pages 1–34, Kigali, Rwanda, 2023. OpenReview.net.

David Morelo. How to use terminal to recover deleted files on mac, 2022. URL `https://www.macgasm.net/data-recovery/recover-deleted-files-with-mac-terminal/`.

OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In Marina Meila and Tong Zhang, editors, *International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763, Virtual, 2021. PMLR.

Navid Rezaei and Marek Z. Reformat. Utilizing language models to expand vision-based commonsense knowledge graphs. *Symmetry*, 14:1715, 2022.

Teven Le Scao, Angela Fan, Christopher Akiki, Elizabeth-Jane Pavlick, Suzana Ili'c, Daniel Hesslow, Roman Castagn'e, Alexandra Sasha Luccioni, Franccois Yvon, Matthias Gallé, and et al. Bloom: A 176b-parameter open-access multilingual language model. *ArXiv preprint arXiv:2211.05100*, 2022.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *IEEE International Conference on Robotics and Automation*, pages 11523–11530, London, UK, 2023. IEEE.

Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Anand Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *ArXiv preprint arXiv:2201.11990*, 2022.

Sai Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. Chatgpt for robotics: Design principles and model abilities. *Microsoft Autonomous Systems and Robotics Research*, 2:20, 2023.

Peter West, Chandra Bhagavatula, Jack Hessel, Jena D. Hwang, Liwei Jiang, Ronan Le Bras, Ximing Lu, Sean Welleck, and Yejin Choi. Symbolic knowledge distillation: from general language models to commonsense models. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4602–4625, Seattle, WA, United States, 2022. Association for Computational Linguistics.

Yunhao Yang, Jean-Raphael Gaglione, Cyrus Neary, and Ufuk Topcu. Automaton-based representations of task knowledge from generative language models. *arXiv preprint arXiv:2212.01944*, 2022.

A Additional Definitions

Product Automata Let $\mathcal{M} \coloneqq \langle Q_{\mathcal{M}}, \Sigma_{\mathcal{M}}, \Gamma_{\mathcal{M}}, \delta_{\mathcal{M}}, \omega_{\mathcal{M}} \rangle$ be a model and let $\mathcal{C} \coloneqq \langle Q, \Sigma, \Gamma, q_0, \delta, \omega \rangle$ be a controller, we define the *product automaton* as an FSA $\mathfrak{P} = \mathcal{M} \otimes \mathcal{C} \coloneqq \langle Q_{\mathfrak{P}}, \delta_{\mathfrak{P}}, q^{\mathfrak{P}}_{init}, \omega_{\mathfrak{P}} \rangle$ as follows:

$$
\begin{aligned}
Q_{\mathfrak{P}} &\coloneqq Q_{\mathcal{M}} \times Q \\
\delta_{\mathfrak{P}}((p,q)) &\coloneqq \{(p',q') \in Q_{\mathfrak{P}} | \delta(q,c,q') = 1 \wedge \delta_{\mathcal{M}}(p,a,p') = 1\} \\
&\quad \text{where } a = \omega(q,\sigma,q') \text{ and } c = \omega_{\mathcal{M}}(p) \\
q^{\mathfrak{P}}_{init} &\coloneqq (p, q_0) \quad \text{where } p \text{ can be any state in } \mathcal{M} \\
\omega_{\mathfrak{P}}((p,q)) &\coloneqq \omega_{\mathcal{M}}(p) \cup \omega(q, \omega_{\mathcal{M}}(p), q') \quad \text{where } q' \in Q \text{ and } \delta(q, \omega_{\mathcal{M}}(p), q') = 1.
\end{aligned}
$$

The trajectories from $\mathfrak{P}$ are in the form $(2^{P_{\mathcal{M}} \cup \overline{P}})^*$, i.e. $\psi_0, \psi_1, \psi_2 \ldots$ where $\psi_i = \omega_{\mathfrak{P}}(q_i, p_i)$.

Planning Domain Definition Language. A PDDL [Haslum et al., 2019] is a formal language used in artificial intelligence and automated planning to define a planning problem. We use PDDL to describe the possible initial states of a problem, the desired goal, and the actions that can be taken to transform the initial state into the goal state. PDDL provides a standardized syntax for specifying a set of predicates—atomic propositions—describing the states of the task, the actions, and the goal specification.

Each action a in PDDL has a name, a *precondition* that must be satisfied before the action can be performed, and an of *effect* that describes how the state of the environment will change after the action is performed. The preconditions and effects are expressed as sets of atomic propositions.

B Algorithms and Grammar Rules

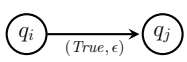
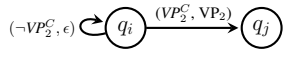
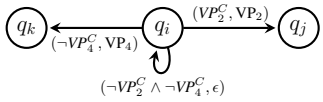
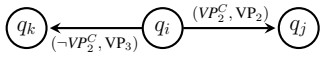
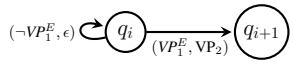
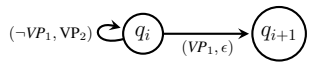
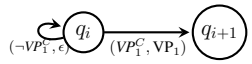
Natural Language Grammar	Formal Representation	Example
VP_1 **and** VP_2	$\text{VP}_1 \wedge \text{VP}_2$	[green light] [and] [no car]
VP_1 **or** VP_2	$\text{VP}_1 \vee \text{VP}_2$	[traffic light] [or] [crosswalk]
no/not VP_1	$\neg\ \text{VP}_1$	[no] [car]
VP_1 **[step number j]**		[go to step] [1]
if VP_1, VP_2. VP_2 **if** VP_1		[if] [green light], [cross]
if VP_1, VP_2. **if** VP_3, VP_4.		[if] [car], [stay]. [if] [no car], [cross].
if VP_1, VP_2 **else** VP_3. VP_2 **if** VP_1, **else** VP_3		[if] [car], [stay], [else] [cross].
wait VP_1 VP_2 VP_2 **after** VP_1		[wait] [green light] [cross]
VP_2 **until** VP_1		[not cross] [until] [green light]
VP_1		[cross road]

Table 2: Rules to convert natural language grammar to formal representations (propositions or FSA transitions). The keywords that define the grammar are in bold.

B.1 Automated Model Extraction from External Knowledge

A model is a transition system that encodes the dynamics of the task environment or the task-relevant knowledge from external knowledge sources. If we obtain an automaton-based model, we can use the model directly for the verification steps. However, in many cases, external knowledge is only available in textual form, such as websites and operation manuals. Again, such textual information is not formally verifiable. To address this problem, we develop an algorithm that automatically constructs a model that encodes the textual information from external sources. The algorithm takes natural language sentences as inputs, extracts propositions corresponding to actions and effects, and returns a transition system representing the model. Note that the algorithm automates the procedure for model construction. However, we do allow users to modify the constructed models to represent additional information.

Extracting Actions and Effects from Text-Based Information Sources Given text-based encodings of externally available knowledge, we apply semantic parsing again to extract all the verb phrases from the text. We consider each verb phrase as an action. Note that some of these verb phrases may have identical meanings, such as "cross the road" and "walk across the road." We align these verb phrases to the output symbols (actions) from the controller by querying GLM in the following format:

```
Do the actions "action 1" and "action 2" lead to the same effect?
 Yes/No.
```

If the response is "yes," we align the action extracted from the textual information to the controller's output symbol. In other words, we consider them as one action. Since we already obtained the effects of the controller's actions (output symbols), we have a list of aligned actions and corresponding effects.

If an extracted action is not aligned with any of the controller's output symbols, we consider it as a new action and query the GLM to obtain the PDDL definition. The result of this process is a list of new actions and the corresponding effects from their PDDL definitions, which we merge with the aligned actions and effects to get a complete list.

Constructing the Model from the Actions and their Effects We design an algorithm named TEXT2MODEL (as presented in Algorithm 1). The algorithm takes the lists of actions and effects as inputs. Recall that the model consists of states, input and output symbols, a transition function, and a label function. The algorithm works as follows:

The algorithm builds 2^M states, where M is the number of atomic propositions in the set of effects. Each state $S \in Q_{\mathcal{M}}$ is associated with a unique label $\omega_{\mathcal{M}}(S)$ that is a conjunction of all the atomic propositions or their negations from the set of effects. For instance, if effects=$\{A, B\}$, then there will be four states with labels $A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B$.

Definition 1. Let $S_1, S_2 \in Q_{\mathcal{M}}$ be the states from the model. We make pairwise comparisons of each atomic proposition in the labels of the two states $\omega_{\mathcal{M}}(S_1)$ and $\omega_{\mathcal{M}}(S_2)$. If only one of the atomic propositions differs, then states S_1, S_2 are considered **neighbors**.

Mathematically, we use the symbol $\oplus$ to denote this elementwise XOR operation. If only one proposition in $\omega_{\mathcal{M}}(S_1) \oplus \omega_{\mathcal{M}}(S_2)$ is evaluated to true, then states S_1, S_2 are considered **neighbors**.

For any two states S_1 and S_2, if there is an action whose effect fills the gap between the labels $\omega_{\mathcal{M}}(S_1)$ and $\omega_{\mathcal{M}}(S_2)$, then the algorithm builds a transition between them with this action as the input symbol. For example, suppose S_1 has a label $A \wedge \neg B$ and S_2 has a label $A \wedge B$, if there exists an action α whose effect is B, then the algorithm builds a transition $\delta(S_1, \alpha) = S_2$. Next, it builds self-transitions with the action "no operation" for every state and builds transitions with the action "no operation" between every two *neighbor* states if no other transition between them already exists.

The "no operation" transitions capture potential environmental changes that are not caused by the controller's behaviors. These transitions provide a conservative approximation of what could potentially happen in the environment. During the verification procedure, these transitions will lead to more failures. We can decide whether to add the "no operation" transitions depending on the stability of the environment and the task requirement.

Algorithm 1: Model Construction from Side Information

procedure TEXT2MODEL(Set[String] *actions*, Set[String] *effects*)
  $Q_{\mathcal{M}}$ = every combination of effects
  $\omega_{\mathcal{M}} := Q_{\mathcal{M}} \to 2^{effects}$
  $\Sigma_{\mathcal{M}}$ = [VP for VP in *actions*]
  $\delta_{\mathcal{M}}$ = []
  for VP in *actions* **do**
    for each pair of states (S_1, S_2) **do**
      Prop = $\omega_{\mathcal{M}}(S_1) \oplus \omega_{\mathcal{M}}(S_2)$ ▷ $\oplus$ is XOR operation
      if $\text{VP}^E \in 2^{\text{Prop}}$ and $\text{VP}^E = \omega_{\mathcal{M}}(S_2)$ over the symbols 2^{Prop} **then**
        $\delta_{\mathcal{M}}$.append($S_1 \times \text{VP} \to S_2$)
      end if
    end for
  end for
  for each pair of neighbor states (S_1, S_2) **do** ▷ Optional
    if there is no transition between S_1, S_2 **then**
      $e = \omega_{\mathcal{M}}(S_1) \oplus \omega_{\mathcal{M}}(S_2)$
      if $e \notin$ *effects* and $\neg e \notin$ *effects* **then**
        $\delta_{\mathcal{M}}$.append($S_1 \times \epsilon \to S_2$)
        $\delta_{\mathcal{M}}$.append($S_2 \times \epsilon \to S_1$)
      end if
    end if
  end for
  for S in $Q_{\mathcal{M}}$ **do**
    $\delta_{\mathcal{M}}$.append($S \times \epsilon \to S$)
  end for
  return $Q_{\mathcal{M}}, \Sigma_{\mathcal{M}}, \delta_{\mathcal{M}}, \omega_{\mathcal{M}}$
end procedure

Algorithm 2: Controller Refinement through Synthesizing from the Model

procedure REFINE(Controller $\mathcal{C}$, Model $\mathcal{M}$, Specification Φ) ▷ $\mathcal{C} = \langle \Sigma, \Gamma, Q, q_0, \delta, \omega \rangle$, $\mathcal{M} = \langle Q_{\mathcal{M}}, \Sigma_{\mathcal{M}}, \delta_{\mathcal{M}}, \omega_{\mathcal{M}} \rangle$
  if $\mathcal{C} \otimes \mathcal{M} \models \Phi$ **then**
    return $\mathcal{C}$
  end if
  find a counter-example $\overline{Q} = \{(p_1, q_1), (p_2, q_2), ... | p_i \in \Sigma_{\mathcal{M}}, q_i \in \Sigma\}$ for $\mathcal{M} \models \neg\Phi$
  convert $\overline{Q}$ to a trajectory $T = \{(\sigma_1 \wedge a_1), ((\sigma_2 \wedge a_2), ... | \sigma_i \in \omega_{\mathcal{M}}, a_i \in \omega)\}$
  Construct a new state $\overline{q}_1$, Q.append($\overline{q}_1$)
  δ.append($(q_{init}, \sigma_1) \to \overline{q}_1$)
  ω.append($(q_{init}, \sigma_1) \to a_1$)
  for i in $[2, n]$ **do**
    Q.append($\overline{q}_i$)
    δ.append($(\overline{q}_{i-1}, \sigma_i) \to \overline{q}_i$)
    ω.append($(\overline{q}_{i-1}, \sigma_i) \to a_i$)
  end for
  δ.append($(\overline{q}_n, True) \to \overline{q}_n$)
  ω.append($(\overline{q}_n, True) \to \epsilon$)
  δ.replace($(q_{init}, \neg c_1 \wedge ... \wedge \neg c_k) \to q_{init}$)
  return $\mathcal{C}$
end procedure

C Additional Empirical Demonstrations

C.1 Cross Road Example

Model Construction After constructing the controller, we want to check whether the knowledge from GPT-4 encoded by the controller is consistent with external knowledge. To do so, we collect information for crossing the road at the traffic light from a tutorial blog [Ali, 2023]. The information is in textual form:

```
1. Reach painted lines on the roadway that indicate a safe crosswalk.
2. Cross the road only when the traffic light turns green.
```

We extract two verb phrases “reach painted lines indicating crosswalk" and “cross the road when the light turns green" from the external textual information. Next, we check whether the extracted verb phrases are aligned with the existing actions by querying GPT-4:

```
Are the actions "reach painted lines indicating crosswalk" and "approach
    pedestrian-crossing" lead to the same effect?
 Yes. Both actions lead to reaching one side of the crosswalk.
Are the actions "cross the road when the light turns green" and "cross
    road" lead to the same effect?
 Yes. Both actions lead to reaching the other side of the road.
```

In practice, we ask for every pair of combinations. However, we only present the aligned pairs for brevity. Note that both verb phrases extracted from the external information are aligned with the controller’s output symbols (actions). Hence the model we will construct shares the same set of vocabularies with the controller.

Since the two verb phrases are aligned with two of the actions from the controller, we directly use the aligned actions and their effects. We send a set of actions {cross road, approach pedestrian-crossing} and a set of corresponding effects {at pedestrian-crossing, at other side of road } as input to Algorithm 1 to construct a model representing the external information, which we present in Figure 2. The `Text2Model` algorithm builds a model that preserves all the possible environment dynamics and possible actions the agent can take. Some of the actions and environment dynamics may be nonsensical and can be manually pruned by users.

C.2 Robot Arm Manipulation

The complete queries and responses:

```
 Task: place all the red blocks off the table.
Environment: there are unknown numbers of red blocks and yellow blocks on
    the table initially. Someone may randomly add a red block or yellow
    block to the table.
Steps for achieving the task:
 1. Target one block on the table.
 2. Classify the color of the targeted block.
 3. If the block is red, place it from the table to an off-table location
    (B). If the block is yellow, leave it on the table.
 4. Go to step 1.

 1. Define an action "target one block" in PDDL.
 Action: target-one-block
 Parameters: ()
 Precondition: (block_on_table)
 Effect: (and (block_targeted))

 2. Define an action "classify the color of the targeted block" in PDDL.
 Action: classify-color
 Parameters: ()
 Precondition: (block_targeted)
 Effect: (or (block_red) (block_yellow))

 3. Define an action "place the block" in PDDL.
 Action: place-block
 Parameters: ()
 Precondition: (and (block_targeted) (block_red))
 Effect: (not (block_targeted))

 3. Define an action "leave the block" in PDDL.
 Action: leave-block
 Parameters: (?b - block ?t - table)
 Precondition: (and (block_targeted) (block_yellow))
 Effect: (not (block_targeted))
```

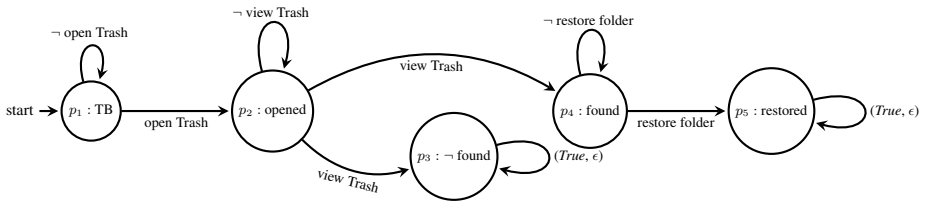


Figure 7: A model that captures information obtained from an external source—a tutorial on restoring files from Trash.

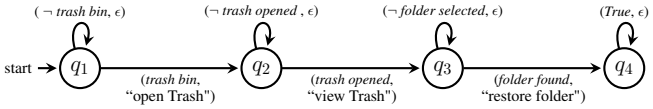


Figure 8: Controllers of the task “restoring a folder from Trash." The top automaton is the original controller constructed from the GLM’s outputs. The middle automaton is the refined controller after the automatic refinement procedure. The bottom automaton is the controller manually refined through interpreting counter-examples.

For model construction, we begin by following Algorithm 1: We send the set of all actions (target block, classify color, leave, place) and the subset of effects (targeted, red, yellow) as the parameters for constructing the model. Then, we manually remove the states that are not reachable by any single action (i.e., the state label is not equal to the effect of any action) and obtain a model as presented in the right figure of Figure 5.

C.3 Computer System Operation

In this example, we demonstrate our algorithms’ capabilities on tasks involving interfacing with a computer system. The task is to restore a folder from the trash bin in the Apple MacOS operating system.

Controller and Model Construction. We start constructing a controller for the task “restoring a folder from the trash in MacBook" by querying GPT-4 and get responses as follows:

```
Steps for restoring one folder from Trash in MacBook:
 1. Open Trash.
 2. View Trash.
 3. Restore folder.

 1. Define an action "Open Trash" in PDDL.
 Action: open-trash
 Parameters: ()
 Precondition: (trash_bin)
 Effect: (trash_opened)

 2. Define an action "View Trash" in PDDL.
 Action: view-trash
 Parameters: ()
 Precondition:  (trash_opened)
 Effect: (or (folder_found) (not (folder_found)) )

 3. Define an action "Restore folder" in PDDL.
 Action: restore-folder
 Parameters: ()
 Precondition:  (folder_found)
 Effect: (folder_restored)
```

We get a list of steps and a set of actions with preconditions from the responses. Then, we follow the grammar and transition rules defined in Table 1 to construct an automaton-based controller, as presented in Figure 8.

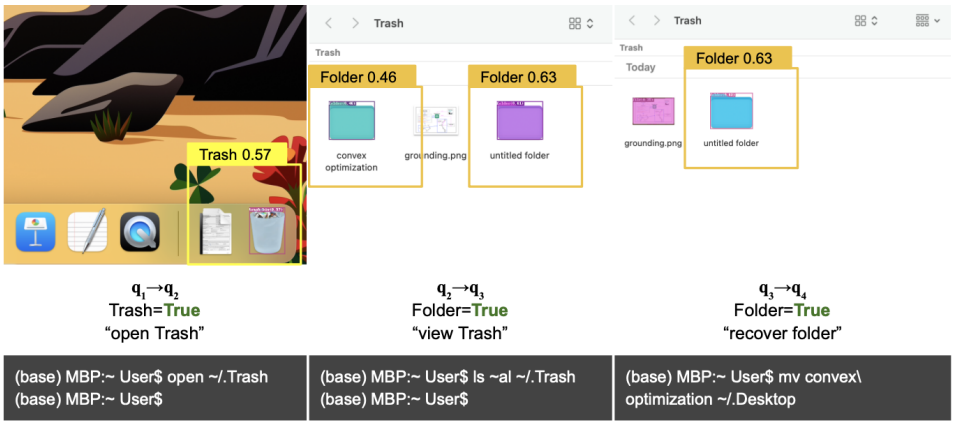


Figure 9: The figure shows the object detection results returned by the Grounded-SAM, the corresponding transitions of the controller we take based on the detection results, and the terminal commands corresponding to the actions outputted by the controller.

After we construct the controller, we want to verify whether this controller is consistent with information from another knowledge source. We collect external information from a MacOS tutorial blog [Morelo, 2022]:

```
1. Navigate to the Trash folder.
2. View the content of the Trash folder.
3. Move a specific folder to your home folder.
```

We extract verb phrases "navigate to Trash folder," "View the content of Trash folder," and "Move a folder to home folder" from the external information. Then, we align these verb phrases to the controller's actions:

```
Are the actions "navigate to Trash folder" and "open Trash" lead to the
    same effect?
 Yes.
Are the actions "Move a folder to home folder" and "recover folder" lead
    to the same effect?
 Yes.
```

We align "navigate to Trash folder" to "open Trash" and "Move a folder to home folder" to "recover folder." Now we have a list of actions ["open Trash," "view Trash," "recover folder"] and their corresponding effects. We build a model accordingly following Algorithm 1. Note that we are not adding "no operation" transitions in this example.

Verification and Refinement. We obtain a product automaton of the controller and the model and verify the product automaton against a specification

$$\Phi = \Diamond(\text{trash bin} \wedge \neg\text{trash opened} \wedge \text{folder found}) \to \Diamond \text{ folder restored}.$$

The model-checking procedure passes, and hence there is no refinement required.

A Step Toward Low-Level Control. We have constructed an automaton-based controller that satisfies the provided specifications. However, the controller only encodes high-level task knowledge. Therefore, we move a step forward to connect high-level knowledge to low-level control implementations, such as to an API or terminal commands that can directly control the system.

We again take advantage of the rich knowledge encoded in the GLM and ask the terminal command for each action from the controller:

```
Give the command line on MacBook for each of the following actions:
1. Open Trash
2. View Trash
3. Restore folder
 1. open ~/.Trash
 2. ls -al ~/.Trash
 3. mv ~/.Trash/folder_name destination_path
```

We then ground the controller to the MacOS by using the Grounded-SAM to perceive the task environment, i.e., images of the OS GUI. Once we determine the transition to take based on the observation, we execute the terminal command corresponding to the action of this transition. This example indicates the potential of our algorithm to connect abstract representations or high-level control logic to system-operable commands. However, this potential has not been fully developed yet, as we currently only use knowledge from the language model to connect high-level actions to operable commands. This approach is incapable if the language model does not encode the knowledge of the operable commands, e.g., unique APIs of a particular robot.