Large Language Models for Verifiable Sequential Decision-Making in Autonomous Systems

Yunhao Yang, Jean-Raphaël Gaglione, Cyrus Neary, Ufuk Topcu
The University of Texas at Austin, United States
{yunhaoyang234, jr.gaglione, cneary, utopcu}@utexas.edu

Abstract: Automaton-based representations of task knowledge play an important role in control and planning for sequential decision-making problems. However, obtaining the high-level task knowledge required to build such automata is often difficult. Meanwhile, *large language models* (LLMs) can automatically generate relevant task knowledge. However, the textual outputs from LLMs cannot be formally verified or used for sequential decision-making. We develop a novel algorithm named GLM2FSA, which constructs a *finite state automaton* (FSA) encoding high-level task knowledge from a brief natural-language description of the task goal. The proposed algorithm thus fills the gap between natural-language task descriptions and automaton-based representations, and the constructed FSAs can be formally verified against user-defined task specifications. We accordingly propose a method to iteratively refine the queries to the LLM based on the outcomes, e.g., counter-examples, from verification. We demonstrate GLM2FSA's ability to build and verify automaton-based representations of everyday tasks and also of tasks that require highly specialized knowledge.

Keywords: Large Language Model, Sequential Decision-Making, Formal Method, Verification

1 Introduction

Automaton-based representations of high-level task knowledge play a key role in planning and learning in sequential decision-making. Such knowledge may include the requirements a designer wants to enforce on an agent or a priori task information about the agent and the environment in which it operates. Automaton-based representations are useful in many applications, such as robot control, reinforcement learning, and program verification.

Despite their utility in a range of applications, capturing high-level task knowledge in automata is not straightforward. Automaton learning algorithms infer such knowledge through queries to a human expert or an automated oracle [1]. In general, these algorithms may require an excessive number of queries to a human, and it is often unclear how an automated oracle can be constructed in the first place. Even in cases in which an oracle exists, either the learning algorithm or the oracle requires prior information, such as the set of possible actions available to the agent and the set of environmental responses, i.e., symbols relevant for the automaton construction. It is often unclear how to obtain this information. Furthermore, the soundness of the inferred automaton depends on the choice of symbols.

We argue—and provide a proof of concept—that recent advances in generative *large language models* (LLMs) can help automatically distill high-level task knowledge into automaton-based representations. Existing LLMs, such as the Generative Pre-trained Transformer series (GPT) of models [2], are capable of generating realistic, human-like text in response to queries. Such text often encodes rich world knowledge. On the other hand, the outputs of LLMs are typically in a textual form that cannot be directly utilized for sequential decision-making or automaton learning. Moreover,

7th Conference on Robot Learning (CoRL 2023), Atlanta, USA.

the textual form outputs are not formally verifiable and hence cannot be used in applications where correctness matters.

We develop an algorithm named *Generative Language Model to Finite State Automaton* (GLM2FSA) to fill the gap between the outputs from LLMs and automaton-based representations of high-level task knowledge. In particular, GLM2FSA produces controllers represented as *finite state automata* (FSAs) from a brief natural-language sentence describing the task (e.g., "cross the road"). It does so by first sending queries containing this task description to a LLM to obtain a list of text instructions organized in steps (and substeps). Then, it parses these textual instructions to define the input and output symbols (i.e., environment propositions and actions) of the FSA. Finally, it interprets each step to construct a corresponding automaton state and its outgoing transitions. GLM2FSA thus constructs FSAs representing *controllers* for sequential decision-making. Figure 1 illustrates the proposed GLM2FSA algorithm, which only takes a brief textual description of the task and outputs a FSA with automatically-defined states, symbols, and transitions.

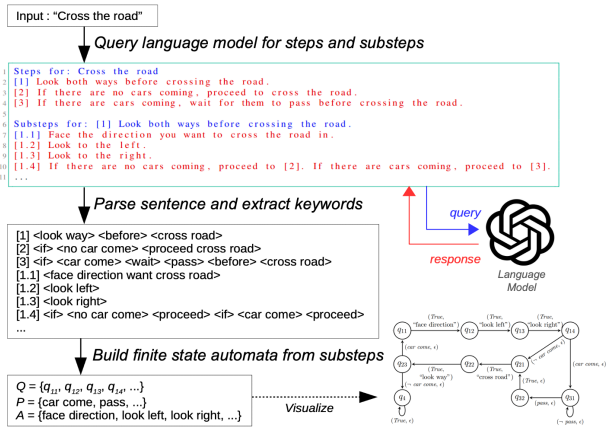


Figure 1: Illustration of the GLM2FSA algorithm.

The FSA-based controllers output by GLM2FSA are formally verifiable against user-defined task specifications. We accordingly propose a method to verify the controllers and to use the results of verification, e.g., counterexamples, as feedback in order to iteratively refine them through additional queries to the LLM. Such systematic verification allows the algorithm to identify and guard against potentially undesirable or nonsensical outputs from the language model, making it a necessary step towards the safe integration of LLMs into automated decision-making systems.

We demonstrate GLM2FSA's capabilities through experimental case studies. To the best of our knowledge, GLM2FSA is the first algorithm that constructs automaton-based representations from textual knowledge extracted from LLMs. It is also the first algorithm to provide an approach to formally verify the knowledge from LLMs in the context of sequential decision-making.

2 Related Work

Extracting Task Knowledge from Language Models. Prior works have studied the extraction of task knowledge from LLMs [3, 4, 5]. However, due to a lack of rich world knowledge, the language model they use cannot generate action plans without providing detailed task descriptions. The recently introduced family of LLMs—GPT [2]—contains rich world knowledge and can generate instructions for a given task [6]. Therefore, some works extract task knowledge by asking GPT for the step-by-step instructions [7, 8]. Meanwhile, a number of recent works have studied how recent advancements in the capabilities of LLMs can be used to extract task-relevant semantic knowledge and to generate plans for task completion in the context of robotics [9, 10, 11]. In contrast to existing works, we are the first to use LLMs to transform natural-language task descriptions into automaton-based representations that can be directly used for sequential decision-making and that can be formally verified against user-defined specifications.

Symbolic Knowledge Representations. Many works focus on constructing symbolic representations of task knowledge from natural language (text) descriptions. Several works [7, 12, 13] extract information from text descriptions of given tasks and construct knowledge graphs for the tasks.

Another work [14] analyzes the causality within the text descriptions and creates causal graphs. In contrast with the existing works, we take advantage of the generative capabilities of LLMs to automatically generate automaton-based representations from brief (one-sentence) textual task descriptions. Such automaton-based representations can be used for formal verification, which the knowledge graphs are not capable of.

Natural Language to Formal Language. Existing works [15, 16, 17] introduce approaches to transform natural language to formal language specifications. Kate et al. [18] induces transformation rules that map natural-language sentences into a formal query or command language. Huang et al. [8] constructs a form of actionable knowledge that machines can recognize and operate on sequentially. However, existing works either cannot operate sequentially or cannot handle conditional transitions, e.g., multiple transitions from one state, which the work we proposed is capable of.

3 Preliminaries

Finite State Automaton. A *finite state automaton* (FSA) is a tuple $\mathcal{A} = \langle \Sigma, A, Q, q_0, \delta \rangle$ where Σ is the input alphabet (the set of input symbols), A is the output alphabet (the set of output symbols), $q_0 \in Q$ is the initial state, and $\delta : Q \times \Sigma \times A \times Q \to \{0, 1\}$ is the transition function, which indicates that a transition exists when it evaluates to 1. Note that we define FSA transitions to be non-deterministic: if an agent is in state $q_i \in Q$ with an input symbol $\sigma \in \Sigma$, the agent can choose the output symbol and next state among the set $\delta(q_i, \sigma) \coloneqq \{(a, q_j) \in A \times Q | \delta(q_i, \sigma, a, q_j) = 1\}$.

We use FSAs in the context of sequential decision-making, where the input alphabet comprises all possible environment observations relevant to the current task. We introduce a set of atomic proposition P such that $\Sigma \coloneqq 2^P$, i.e., an input symbol $\sigma \in \Sigma$ is the set of atomic propositions in P that evaluate to *True*. We also introduce a set of atomic propositions P_A for the output alphabets $A \coloneqq 2^{P_A}$, and we allow for a "no operation/empty" symbol $\epsilon \in A$.

In this work, we use FSAs output by the proposed algorithm to represent *controllers*, which are system components responsible for making decisions and taking actions based on the system's state. We refer to the input and output alphabets (Σ and A) of the FSA as the *condition set* and *action set*, where $\sigma \in \Sigma$ and $a \in A$ represent conditions and actions.

Semantic Parsing. Semantic parsing is a task in *natural language processing* (NLP) that converts a natural language utterance to a logical form: a machine-understandable representation. We follow the approach that predicts part-of-speech (POS) tags for each token and that builds *phrase structure* depending on *phrase structure rules*, also known as a *grammar*. POS tags include noun (N), verb (V), adjective (AJD), adverb (ADV), etc. Phrase structures are a tree-structured logical form whose leaves are the POS tags of the given natural language utterance (i.e., sentence). Phrase structure rules organize the POS tags into phrases like noun phrases (NP) and verb phrases (VP).

Definition 1. A *Noun Phrase (NP)* is a group of words headed by a noun. A *Verb Phrase (VP)* is composed of a verb and its arguments. VP follows the following grammars:

$$VP \longleftarrow V\ VP \text{ or } VP \longleftarrow V\ NP.$$

The left-hand side of the grammar is composed of the components on the right-hand side. The grammar defines a verb phrase as being either composed of a verb and another verb phrase or composed of a verb and a noun phrase.

To standardize the words under the phrase structure, the parsing approach converts all the words to their original form, e.g., it removes singular or plural, past tense, etc. This operation eliminates cases where phrases with the same words in different tenses are categorized as being distinct.

Algorithm 1: Natural Language to FSA

procedure STEP2FSA(function keyword_handler, List[String] STEPS, List[String] keywords)
  Q = [a state for each step] + [absorbing state] ▷ define states, a state represents a step
  $q_0 = Q[0]$ ▷ define the initial state
  $P, A, \delta = \{\}, \{\epsilon\}, \{\}$ ▷ define input and output symbols and transitions
  for state_number in $[0 : |Q| - 1]$ **do**
    S_NUM = step number of the current state
    VP^C, VP^A, KEYS = **parse**(STEPS[S_NUM])
    if any(keywords) in KEYS **then**
      keyword_handler(Q, VP^C, VP^A, KEYS, keywords)
    else
      create $\delta(q_{\text{S\_NUM}}, True, \text{VP}^A, q_{\text{S\_NUM}+1})$ ▷ create a transition
      $\Sigma, A := \Sigma \cup VP^C, A \cup VP^A$ ▷ add input and output symbols
    end if
  end for
  return P, A, Q, q_0, F, δ
end procedure

4 Methodology

We propose an algorithm named *Generative Language Model to Finite State Automaton* (GLM2FSA). The algorithm first uses a natural-language description of the task of interest to query the LLM and to obtain step-by-step task instructions in textual form. It then automatically parses these text-based instructions to construct a controller, represented as an automaton, which can be used for sequential decision-making.

Extracting Textual Knowledge. The first step of the proposed approach is to distill task-relevant textual knowledge from the LLM by iteratively prompting it with structured natural-language queries. Given a task description of interest, the algorithm asks for steps to achieve the task. We also provide a method to refine a step (or a substep) into its constituent substeps. The input prompts follow the format below (prompts sent to the LLM in blue, completion of the LLM in red):

```
Steps for: task description
[1] step description
[step number] step description
...
Substeps for: [step number] step description
[step number.1] substep description
[substep number] substep description
...
```

This iterative querying process allows for the automated decomposition of the task description into a structured hierarchy of steps and substeps up to a pre-specified depth. Alternatively, it could be used as a mechanism for the refinement of the LLM's output: steps can be automatically broken into substeps if they are unclear or too difficult to accomplish without further guidance. This information for step refinement could come, for example, from a downstream verification algorithm or even from a human operator.

Building FSA from Textual Knowledge. The next step in the proposed approach is parsing textual LLM outputs and constructing automata-based controllers. Algorithm 1, which we refer to as GLM2FSA, transforms step descriptions from textual form to finite state automata.

The algorithm first applies semantic parsing to each step description to obtain the keywords and verb phrases that it contains. These keywords belong to a pre-defined set of words that we use to define our grammar for automaton construction, such as *if* and *wait*. Furthermore, each verb phrase output by the algorithm's semantic parsing step is interpreted either as a condition or as an action. We define these two categories of verb phrases more precisely as follows:

Definition 2. VP^A is a verb phrase that leads to actions, and VP^C is a verb phrase indicating the conditions for triggering the transitions.

For each verb phrase, the algorithm classifies it as VP^A by default, unless the grammar associated with the keywords specifies that it is a VP^C. The algorithm adds VP^A to the set of output symbols A and VP^C to the set of atomic propositions P. In Algorithm 1, we refer to the process that executes this keyword and verb phrase extraction as the **parse** function.

Category	Grammar	Transition Rule	Example
Default Transition	VP^A	$q_i \xrightarrow{(True,\, \text{VP}^A)} q_{i+1}$	[dial number]
Direct Transition	VP^A [j]	$q_i \xrightarrow{(True,\, \epsilon)} q_j$	[proceed] [1]
Conditional Transition	**if** VP^C, VP^A VP^A **if** VP^C	$(\neg VP^C, \epsilon)$ ↺ $q_i \xrightarrow{(VP^C,\, \text{VP}^A)} q_j$	[if] [no car], [cross]
Self Transition	**wait** VP^C VP^A VP^A **after** VP^C	$(\neg VP^C, \epsilon)$ ↺ $q_i \xrightarrow{(VP^C,\, \text{VP}^A)} q_{i+1}$	[wait] [car pass] [cross]

Table 1: Transition rules defined for keywords under a specific grammar.

Next, the algorithm constructs a FSA from the parsed steps and the verb phrases within these steps. For each step, the algorithm adds a state q_i representing the current step i to Q. The algorithm defines the state corresponding to the first step as the initial state. It also adds an absorbing state after the state corresponding to the final step. The absorbing state only has one self-transition with input *True* and output "no operation". Finally, the algorithm builds transitions between the states using the various transition rules that we define in Table 1, and detail further in the appendix.

5 Verification and Refinement

Automaton-based representations of knowledge are compatible with existing methods for formal verification against task specifications. We present an approach to apply such computational techniques to `GLM2FSA`'s outputs. Given an automaton-based controller $\mathcal{C}$ output by `GLM2FSA`, we use a *model* $\mathcal{M}$ to verify the behavior of $\mathcal{C}$ against some task specifications of interest. We assume the model and specifications are provided from externally available knowledge sources.

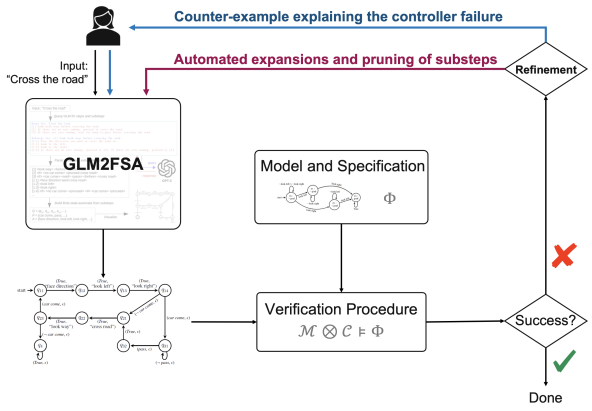


Figure 2: Illustration of the proposed procedure for automated verification and refinement. We verify the automaton generated by `GLM2FSA`, and we send new queries to the LLM to refine the automaton if this verification fails.

Verifying the Automaton-Based Controllers Against Models and Task Specifications. We begin with the problem of automatically checking that the behaviors of the controllers satisfy desired specifications when verified against a *model*. A *model* is a representation of any a priori task-related knowledge provided by the user, e.g., an abstract model of how the task environment responds to the actions taken by the controller. The verification procedure checks if the controller is consistent with this existing knowledge or if it will satisfy logical-based specifications

when implemented against the model. Such systematic verification is necessary to identify and guard against undesirable or nonsensical outputs from the LLM.

We define the *model* as $\mathcal{M} := \langle \Sigma_{\mathcal{M}}, \Gamma_{\mathcal{M}}, Q_{\mathcal{M}}, p_0, \delta_{\mathcal{M}}, \lambda_{\mathcal{M}} \rangle$. $\Sigma_{\mathcal{M}} := A = 2^{P_A}$ is a set of input symbols, where P_A is a set of atomic propositions representing actions in the controller. $\Gamma_{\mathcal{M}} := 2^{\{goal\} \cup P}$ is a set of output symbols, where P is the set propositions from $\mathcal{C}$ and *goal* is a special proposition. $Q_{\mathcal{M}}$ 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, $p_0 \in Q_{\mathcal{M}}$ is an initial state, and $\lambda_{\mathcal{M}} : Q_{\mathcal{M}} \to \Gamma_{\mathcal{M}}$ is a labeling function.

We use *linear temporal logic* (LTL) [19] to define task specifications Φ that the controller $\mathcal{C}$ should satisfy, given the model $\mathcal{M}$. LTL is a formal language that expresses system properties that evolve over time. It is built on top of propositional logic by extending it with temporal operators—$\Diamond$ ("eventually") and $\Box$ ("always")—which allow for reasoning about the system's future behavior. We provide a formal definition of LTL in the appendix, and we refer to Baier and Katoen [20] for further details.

We define specifications Φ over atomic propositions in $P \cup P_A \cup \{goal\}$, and evaluate them over trajectories in the form $(2^{P \cup P_A \cup \{goal\}})^*$. In the context of our verification problem, specifications Φ represent desired outcomes that the controller should satisfy, given some assumptions on the properties of the system that the controller interacts with.

To verify that the controller $\mathcal{C}$ satisfies the specification Φ given the model $\mathcal{M}$, we solve the following automated verification problem,

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

where $\mathcal{M} \otimes \mathcal{C}$ denotes the so-called *product automaton* describing the interactions of the controller $\mathcal{C}$ with the model $\mathcal{M}$, which we define formally in the appendix. We leave the details of the automated verification problem to Baier and Katoen [20]. In this work, we use the NuSMV model checker [21] for this purpose. The output of the automated verification problem is binary: the automaton-based controller $\mathcal{C}$ either satisfies the specification Φ given the model or it does not. We present a detailed example of how we use the model checker to verify an example controller in the appendix.

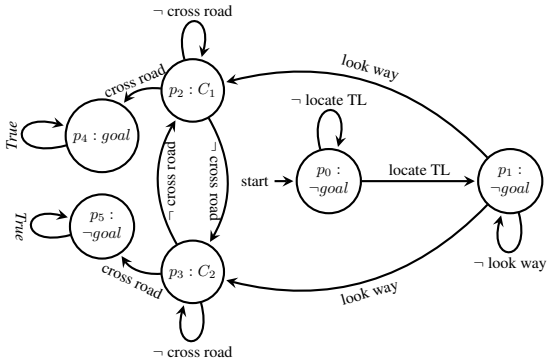
Finally, we note that due to the stochastic nature of generative models, the LLM may often output different phrases to represent the same concept. Hence each action may correspond to multiple verb phrase expressions, which could cause problems during the automated verification procedure. To ensure that the verification does not fail due to an inability to recognize synonyms, we directly query the LLM to ask if any two verb phrases refer to the same action.

The verification step can provide formal guarantees on whether or not the controller satisfies the provided specifications. If the controller fails the verification step, the model checker will return a sequence of states from the product automaton indicating exactly how the specification was failed. Such counter-examples allow users to interpret the reasons behind failures and to modify the controller accordingly. If the automaton output by `GLM2FSA` fails to satisfy the provided task specification, we provide two approaches to refine the controller $\mathcal{C}$.

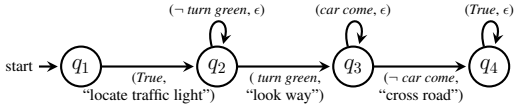
Counterexample-Guided Controller Refinements. The first approach asks the user to manually modify the input prompt to the LLM and to use the resulting outputs to update the controller. As previously described, if the verification steps fails, the model checker generates a counter-example. The user can then use this information to modify the LLM's input prompt in a way that addresses the issue observed in the counter-example. After obtaining the LLM's outputs, we apply `GLM2FSA` again to construct the updated controller.

Automated Refinement Through Substep Expansions. The second approach queries the LLM for substeps to automatically refine the controller $\mathcal{C}$. We can then again solve the verification problem for the newly-updated $\mathcal{C}$. This leads to an iterative process that uses the information provided by the verification step to improve the distilled automaton.

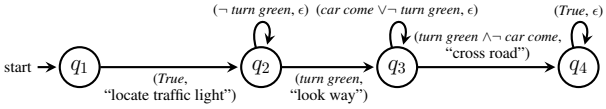
During each refinement step, we apply Algorithm 2 (included in the appendix) to query the LLM for the next-layer steps (DEPTH = DEPTH + 1). This expands each of the automaton's transitions into



(a) A model $\mathcal{M}$ for "cross the road." PC stands for Pedestrian Crossing, C_1 = green $\wedge\neg$ car come $\wedge\neg$ goal, and C_2 = (car come $\vee\neg$ green) $\wedge\neg$ goal. We annotate each node as "state: label."



(b) The FSA for the task "cross the road at the traffic light."



(c) The final FSA after the manual refinement.

Figure 3: The iterations of refinements. The user manually queries GPT-3 to refine the steps based on the counterexample from the verification.

more detailed representations, describing its necessary substeps. We refer to the state that represents the beginning of the transition to be expanded as the *parent state* and the states that represent that transition's substeps as the *child states*. We continue the loop of expanding the automaton's transitions and applying automated verification until all the specifications are satisfied or the maximum number of layers is reached. This maximum number of layers is a user-defined constant. If we reach the maximum number of layers and still cannot satisfy the specifications, we consider the task to be unrepresentable by an automaton.

To prevent the above iterative refinement procedure from generating unnecessarily large automaton-based controllers, we also design a pruning process. This pruning process proceeds as follows: 1) start from the deepest-layer steps and replace the first set of children states with their parent state, 2) check if the controller still satisfies all the specifications, 3) keep the controller as it is if the specifications are satisfied, otherwise add the children states back, and 4) continue steps 1 to 3 for all of the children states at each level of the hierarchy.

6 Experimental Results

Controller construction. To demonstrate the application of `GLM2FSA` to a relatively simple commonsense task, we begin with the task prompt "cross the road at the traffic light." We present three additional examples, which include substep expansions and controllers designed for more complex, domain-specific tasks, in the appendix. In all of the experiments, we use the *text-davinci-003* model from the GPT-3 model family [2] as the LLM queried by `GLM2FSA`.

First, we apply the algorithm to construct a FSA for step descriptions. In this example, we query the LLM for the text-based steps of crossing the road at a traffic light. The queries and the responses from the LLM are as follows:

```
Steps for: Cross the road at the traffic light
[1] Locate the traffic light.
[2] Wait for the traffic light to turn green.
[3] Look both ways before crossing the road.
[4] Cross the road if no cars are coming.
```

GLM2FSA constructs a FSA to represent these steps, illustrated in Figure 3b. The constructed FSA successfully represents all the required knowledge, including actions and conditions. The task steps are represented by states in the FSA. The algorithm creates the set of atomic propositions $P = \{car\ come, turn\ green, traffic\ light\}$ and output propositions $P_A = \{$ "look way", "cross road", "locate traffic light", $\epsilon\}$ from the extracted verb phrases. Figure 3b indicates that GLM2FSA is capable of building automaton-based representations that are unambiguous and able to represent task-relevant knowledge in step descriptions.

Verification and Refinement. We begin by verifying the correctness of the controller $\mathcal{C}$ from Figure 3b using the model $\mathcal{M}$ illustrated in Figure 3a. We define the specification as $\Phi =$ traffic light $\wedge \square \lozenge($ green $\wedge \neg$ car come$) \rightarrow \lozenge\, goal$ (if the agent is at a traffic light and there always will eventually be a time when the traffic light is green and no car is coming, then it should eventually reach the goal).

In this example, the controller fails the verification step with a counter-example of state sequences $p_0 \rightarrow p_1 \rightarrow p_3 \rightarrow [\text{infinite loop } p_5]$. State p_5 in $\mathcal{M}$ is reached because it is possible for the controller to take action "cross road" when the traffic light is red. This mistake happens in scenarios where the traffic light switches from green to red while the agent is waiting for cars to stop coming. This is a potentially dangerous edge case that the LLM fails to consider. We emphasize that this edge case could easily be missed by a human as well. It is only by formally verifying the possible behaviors of the system against the model that the potential problem becomes apparent.

Once the model checker returns a counterexample, we can use the information from the counterexample to manually refine the controller. To handle the above corner case, we simply need to ensure that the traffic light is green and that there are simultaneously no cars coming before taking the "cross road" action. So, to address the issue, the user can modify the input prompt as follows:

```
Refine the following steps to ensure the action "cross the road" is performed
     under conditions "traffic light turns green" and "no cars are coming":
[1] Locate the traffic light.
...
[4] Cross the road if no cars are coming.
[1] Locate the traffic light.
[2] Wait for the traffic light to turn green.
[3] Look both ways before crossing the road.
[4] Cross the road if no cars are coming and the traffic light is green.
```

We then apply the algorithm GLM2FSA again to the refined responses and construct a refined controller. We present the refined controller in Figure 3c. Now, the controller passes all the verification steps, and hence it is finalized.

7 Conclusions

We provide a proof-of-concept for the automatic construction of automaton-based representations of abstract task knowledge from LLMs. We propose an algorithm, GLM2FSA, that accepts brief natural-language descriptions of tasks as input, queries a LLM, and then constructs an automaton from the language model's responses. The algorithm is highly automated, requiring only a short task description to build machine-understandable knowledge representations. We additionally propose methods to formally verify the automata produced by GLM2FSA, and to use the results of verification to iteratively refine the inputs to the LLM. Experimental results demonstrate the capabilities of GLM2FSA: The generated automaton-based controllers capture task-relevant knowledge, even when the relvant keywords for controller construction is not included in the original input prompt.

References

[1] K. S. Narendra and M. A. L. Thathachar. Learning automata - a survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 4(4):323–334, 1974. doi:10.1109/TSMC.1974.5408453.

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[3] W. Xiong, J. Du, W. Y. Wang, and V. Stoyanov. Pretrained encyclopedia: Weakly supervised knowledge-pretrained language model. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=BJlzm64tDH.

[4] J. Davison, J. Feldman, and A. M. Rush. Commonsense knowledge mining from pretrained models. In *Conference on Empirical Methods in Natural Language Processing*, pages 1173–1178, 2019. doi:10.18653/v1/D19-1109. URL https://doi.org/10.18653/v1/D19-1109.

[5] F. Petroni, T. Rocktäschel, S. Riedel, P. Lewis, A. Bakhtin, Y. Wu, and A. Miller. Language models as knowledge bases? In *Conference on Empirical Methods in Natural Language Processing*, pages 2463–2473, 2019.

[6] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=d7KBjmI3GmQ.

[7] P. West, C. Bhagavatula, J. Hessel, J. D. Hwang, L. Jiang, R. L. Bras, X. Lu, S. Welleck, and Y. 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, 2022. doi:10.18653/v1/2022.naacl-main.341. URL https://doi.org/10.18653/v1/2022.naacl-main.341.

[8] W. Huang, P. Abbeel, D. Pathak, and I. 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, 2022.

[9] S. Vemprala, R. Bonatti, A. Bucker, and A. Kapoor. ChatGPT for robotics: Design principles and model abilities, 2023. Published by Microsoft.

[10] B. Ichter, A. Brohan, Y. Chebotar, C. Finn, K. Hausman, A. Herzog, D. Ho, J. Ibarz, A. Irpan, E. Jang, R. Julian, D. Kalashnikov, S. Levine, Y. Lu, C. Parada, K. Rao, P. Sermanet, A. Toshev, V. Vanhoucke, F. Xia, T. Xiao, P. Xu, M. Yan, N. Brown, M. Ahn, O. Cortes, N. Sievers, C. Tan, S. Xu, D. Reyes, J. Rettinghouse, J. Quiambao, P. Pastor, L. Luu, K. Lee, Y. Kuang, S. Jesmonth, N. J. Joshi, K. Jeffrey, R. J. Ruano, J. Hsu, K. Gopalakrishnan, B. David, A. Zeng, and C. K. 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, 2022.

[11] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, P. Sermanet, T. Jackson, N. Brown, L. Luu, S. Levine, K. Hausman, and B. Ichter. Inner monologue: Embodied reasoning through planning with language models. In *Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pages 1769–1782, 2022.

[12] N. Rezaei and M. Z. Reformat. Utilizing language models to expand vision-based commonsense knowledge graphs. *Symmetry*, 14:1715, 2022.

[13] M. He, T. Fang, W. Wang, and Y. Song. Acquiring and modelling abstract commonsense knowledge via conceptualization. *arXiv preprint arXiv:2206.01532*, 2022.

[14] Y. Lu, W. Feng, W. Zhu, W. Xu, X. E. Wang, M. Eckstein, and W. Y. Wang. Neuro-symbolic procedural planning with commonsense prompting. *arXiv preprint arXiv:2206.02928*, 2022.

[15] C. Baral, J. Dzifcak, M. A. Gonzalez, and J. Zhou. Using inverse lambda and generalization to translate english to formal languages. In *International Conference on Computational Semantics*, pages 35–44, 2011. URL https://aclanthology.org/W11-0105/.

[16] D. Sadoun, C. Dubois, Y. Ghamri-Doudane, and B. Grau. From natural language requirements to formal specification using an ontology. In *International Conference on Tools with Artificial Intelligence*, pages 755–760, 2013. doi:10.1109/ICTAI.2013.116. URL https://doi.org/10.1109/ICTAI.2013.116.

[17] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner. ARSENAL: automatic requirements specification extraction from natural language. In *NASA Formal Methods*, volume 9690 of *Lecture Notes in Computer Science*, pages 41–46, 2016. doi:10.1007/978-3-319-40648-0_4. URL https://doi.org/10.1007/978-3-319-40648-0_4.

[18] R. J. Kate, Y. W. Wong, and R. J. Mooney. Learning to transform natural to formal languages. In *National Conference on Artificial Intelligence*, pages 1062–1068, 2005. URL http://www.aaai.org/Library/AAAI/2005/aaai05-168.php.

[19] A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science*, pages 46–57, 1977. doi:10.1109/SFCS.1977.32. URL https://doi.org/10.1109/SFCS.1977.32.

[20] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[21] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. 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, 2002. doi:10.1007/3-540-45657-0_29. URL https://doi.org/10.1007/3-540-45657-0_29.

A Additional Background and Definitions

Large Language Models. A *large language model* (LLM) produces human-like text *completion* from a given initial text (*prompt*). The produced texts continue filling the content from that prompt. Recent LLMs are deep learning models with millions or billions of parameters.

Generative Pre-trained Transformer Series are the current state-of-the-art LLMs. We use GPT-3 in our experiments. It is pretrained by five large-scale datasets with over 5 billion words. GPT-3 offers four primary models with different capabilities for different tasks [2]. *Davinci-003* is the most capable model that can do question-answering, next-sentence prediction, and text insertion. We query *Davinci-003* to obtain task instructions for empirical analysis.

GPT-3 allows users to customize settings by setting the hyper-parameters. For instance, *max_tokens* restricts the maximum number of *tokens* (words and punctuation) of the generated text, and *temperature* defines the randomness of the outputs. We propose an algorithm that specifies grammar rules for certain keywords. Hence we set *bias* on the keywords to ensure the model outputs them instead of their alternations. Setting bias to keywords eliminates the need to transform synonyms to the corresponding keywords or define new rules for those synonyms.

Linear Temporal Logic. Formally, LTL formulas are defined inductively as: $\varphi := p \in P_A \mid \neg\varphi \mid \varphi \vee \varphi \mid \circ\varphi \mid \varphi \,\mathbf{U}\, \varphi$ Intuitively, an LTL formula consists of

- A set of atomic propositions, denoted by lowercase letters (e.g., car come), represent the system's state.
- A set of temporal operators describes the system's temporal behavior.
- A set of logical connectives, such as negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$), that can be used to combine atomic propositions and temporal operators.

As syntax sugar, along with additional constants and operators used in propositional logic, we allow the standard temporal operators $\Diamond$ ("eventually") and $\Box$ ("always").

Product Automaton. Let a controller be $\mathcal{C} := \langle \Sigma, A, Q, q_0, \delta \rangle$ with input alphabet $\Sigma := 2^P$, output alphabet $A := 2^{P_A}$, and non-deterministic transition function $\delta : Q \times \Sigma \times A \times Q \to \{0, 1\}$.

Let a model be a tuple $\mathcal{M} := \langle \Sigma_{\mathcal{M}}, \Gamma_{\mathcal{M}}, Q_{\mathcal{M}}, p_0, \delta_{\mathcal{M}}, \lambda_{\mathcal{M}} \rangle$ with input alphabet $\Sigma_{\mathcal{M}} = A$, output alphabet $\Gamma_{\mathcal{M}} = 2^{P \cup \{goal\}}$, a non-deterministic transition function $\delta_{\mathcal{M}} : Q_{\mathcal{M}} \times \Sigma_{\mathcal{M}} \times Q_{\mathcal{M}} \to \{0, 1\}$, and a label function $\lambda_{\mathcal{M}} : Q_{\mathcal{M}} \to \Gamma_{\mathcal{M}}$.

We define the *product automaton* as a transition system $\mathfrak{P} = \mathcal{M} \otimes \mathcal{C} := \langle Q_{\mathfrak{P}}, \delta_{\mathfrak{P}}, q^{\mathfrak{P}}_{init}, \lambda_{\mathfrak{P}} \rangle$ as follows:

$$
\begin{aligned}
Q_{\mathfrak{P}} &:= Q_{\mathcal{M}} \times Q \\
\delta_{\mathfrak{P}}((p, q)) &:= \{(p', q') \in Q_{\mathfrak{P}} | \delta(q, \lambda_{\mathcal{M}}(p) \cap \Sigma, a, q') = 1 \text{ and } \delta_{\mathcal{M}}(p, a, p') = 1, \text{for some } a \in A\} \\
q^{\mathfrak{P}}_{init} &:= (p_0, q_0) \\
\lambda_{\mathfrak{P}}((p, q), (p', q')) &:= \{\lambda_{\mathcal{M}}(p) \cup a | a \in A \text{ and } \delta(q, \lambda_{\mathcal{M}}(p) \cap P, a, q') = 1 \text{ and } \delta_{\mathcal{M}}(p, a, p') = 1\}
\end{aligned}
$$

Here, $\delta_{\mathfrak{P}} : Q_{\mathfrak{P}} \to 2^{Q_{\mathfrak{P}}}$ is a non-deterministic transition function, and $\lambda_{\mathfrak{P}} : Q_{\mathfrak{P}} \times Q_{\mathfrak{P}} \to 2^{P \cup P_A \cup \{goal\}}$ is a label function. The product automaton generates infinite trajectories $(p_0, q_0), (p_1, q_1), \ldots$ by beginning in an initial state $q^{\mathfrak{P}}_{init}$ and following the nondeterministic transition function $\delta_{\mathfrak{P}}$ thereafter. Labeled trajectories are then generated by applying the labeling function $\lambda_{\mathfrak{P}}$ to these trajectories within the product automaton, i.e. $\psi_0\psi_1, \ldots \in (2^{P \cup P_A \cup \{goal\}})^*$ where $\psi_i \in \lambda_{\mathfrak{P}}((p_i, q_i), (p_{i+1}, q_{i+1}))$. When using the product automaton to solve the model-checking problem from Equation (1), we check that all possible labeled trajectories generated by the product automaton belong to the language defined by the LTL specification.

Algorithm 2: Query the LLM for Task Instructions

```
procedure GLM2STEP(String TASK_DESC, integer DEPTH, List[String] keywords)  ▷ Obtain the instructions for a given task; depth indicates how detailed the instructions are.
    GLM.bias = keywords
    PROMPT = "Steps for: " + TASK_DESC + "\n [1]"
    ANSWER = GLM(PROMPT)
    STEP_NUMBERS = ["[1]", "[2]",...]
    for i in range(1, DEPTH) do
        SUB_NUMBERS = []
        ANSWER = []
        for number in STEP_NUMBERS do
            SUB_PROMPT = "Substeps for "+number
            ANSWER.append(GLM(SUB_PROMPT))
            SUB_NUMBERS.append("[1.1]",...)
        end for
        STEP_NUMBERS = SUB_NUMBERS
    end for
    return STEPS = (STEP_NUMBERS, ANSWER)
end procedure
```

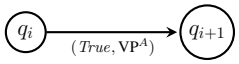
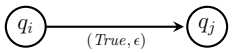
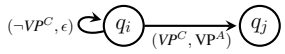
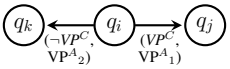
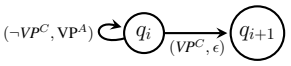
Category	Grammar	Transition Rule	Example
Default Transition	VP^A		[dial number]
Direct Transition	VP^A [j]		[proceed] [1]
Conditional Transition	**if** VP^C, VP^A VP^A **if** VP^C		[if] [no car], [cross]
Conditional Transition (if else)	**if** VP^C, $\text{VP}^A{}_1$. **if** $\neg$ VP^C, $\text{VP}^A{}_2$ **if** VP^C, $\text{VP}^A{}_1$ **else** $\text{VP}^A{}_2$ $\text{VP}^A{}_1$ **if** VP^C, **else** $\text{VP}^A{}_2$		[if] [no car], [cross]. [if], [car] [stay].
Self Transition	**wait** VP^C VP^A VP^A **after** VP^C		[wait] [car pass] [cross]
	VP^A **until** VP^C		[stay] [until] [car pass]

Table 2: Transition rules defined for keywords under a specific grammar.

B Additional Explanations on `GLM2FSA`

Algorithm 2 depicts an iterative process that first queries the LLM for steps to accomplish the task description and subsequently for substeps to accomplish these individual steps.

Table 2 shows the grammar rules and their corresponding transition rules. Note that if a verb phrase VP includes one or more of the words *and*, *or*, *no*, or *not*, then the algorithm refines VP as follows:

- no/not VP_1 = $\neg$ VP_1,
- VP_1 and VP_2 = $\text{VP}_1 \wedge \text{VP}_2$,
- VP_1 or VP_2 = $\text{VP}_1 \vee \text{VP}_2$.

C Additional Explanations on Transition Rules

Default Transitions. We define default transitions as transitions from the current state to the next state with condition $True$. Each state q_i only has one outgoing transition to its next state q_{i+1}, with

the verb phrases from the i^{th} step as the output symbols. A default transition $\delta(q_i, True, \text{VP}_i^A, q_{i+1})$ exists, unconditionally of the valuation of the atomic propositions in P (hence the condition $True$). A demonstration of the default transition is presented in the first row of Table 2.

Occasionally, the algorithm replaces the default transitions with special transitions defined by a grammar over the keywords, which we define in Table 2.

Direct State Transitions. We define a transition from the current state q_i to a state other than the next state q_{i+1}, called a direct state transition. The direct state transition happens when there is a verb phrase in the step description consisting of the number corresponding to another step. The rule for direct state transitions is presented in Table 2.

The algorithm builds a direct state transition from the current state to the state representing step j with output symbol ϵ (no operation).

Conditional Transitions. We define a type of transition named conditional transition, in which the transition only happens when certain conditions are satisfied. The condition of a conditional transition is a conjunction of one or more atomic propositions in P. The conditional transition is caught by the keyword 'if'. The transition rule for conditional transition is defined in the third row of Table 2.

The algorithm builds two transitions from each sentence with the above patterns. The first transition consists of a starting state q_i, a conjunction of atomic propositions VP^C, a target state q_j, and a set of outputs VP^A. The second transition is a self-transition at q_i with a condition $\neg\text{VP}^C$. The second transition does not have any output.

If the VP^A does not lead to a direct state transition, the first transition ends at q_{i+1}.

Self-Transitions. We define a type of transition called self-transition, whose starting and target states are identical. The algorithm builds two transitions: a self-transition whose starting and target states are both the current state, and a second transition that originates from the current state and ends at the next state. The algorithm triggers self-transition when observing a verb phrase containing the keywords 'wait', 'after', or 'until'. The rules for building the two transitions are defined in the fourth row in Table 2.

D Additional Examples of Controller Construction

Phone Call Example: Hierarchical Expansions of Automaton Substeps. We have shown the capability of `GLM2FSA` to generate first-layer step descriptions. In this example, we explore generating second-layer descriptions as well, and integrating the descriptions from the two layers.

In practice, we do not need the details for all the steps generated by the LLM. Some step descriptions are straightforward, while others may need further explanation. Therefore, we apply `GLM2FSA` to build a *finite state automaton* (FSA) that represents the first-layer and the second-layer step descriptions simultaneously.

First, we apply the algorithm to query the LLM and obtain the step and substep descriptions:

```
Steps for: Make a phone call with a cellphone
[1] Find the phone number you want to call.
[2] Dial the number on your cellphone.
[3] Press the call button.
[4] Wait for the person to answer the phone.

Substeps for: [2] Dial the number on your cellphone
[2.1] Enter the country code.
[2.2] Enter the area code.
[2.3] Enter the 7-digit phone number.
```

Note that in this example we only request substeps for the second step, whose action is "dial number", while the other steps are already straightforward.

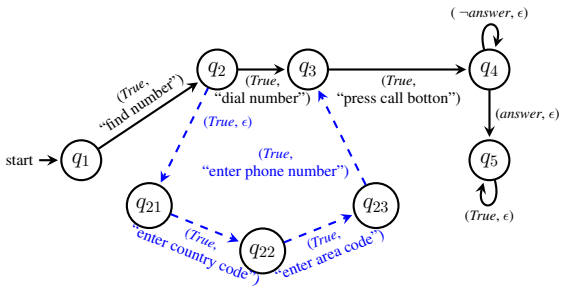


Figure 4: Automaton-based representation for "making a phone call with a cellphone". Transitions in black solid lines show the first-layer steps. Transitions in blue dashed lines show the FSA the steps associated with q_2 is extended to its substeps for more details.

The algorithm builds two FSAs representing the first-layer steps and the second-layer substeps, respectively. Then, we remove the outgoing transition from the state q_2 corresponding to the second step. Instead, we build an outgoing transition from q_2 to the initial state of the FSA for second-layer substeps and replace the transition to the final state with the transition to state q_3. Hence we create a partially extended FSA.

This example indicates that `GLM2FSA` can construct partially scalable FSAs. Every state in the constructed FSA can be extended to represent more details.

Secure Multi-Party Computation: Distilling Domain-Specific Task Knowledge. In addition to the daily-life tasks presented above, we show that `GLM2FSA` can also devise controllers for domain-specific tasks, which require highly specialized task knowledge. Such examples expand the application of `GLM2FSA` to fields where human expertise would previously have been required.

In particular, we consider the task of "secure multi-party computation". Secure multi-party computation (MPC) is a technique that allows multiple parties to jointly compute a function on their private inputs without revealing anything about their inputs to each other or to any other third party. MPC is a specialized problem in computer security, which may not be well-known by people outside this field.

We query LLM and get responses as the following:

```
Steps for: secure multi-party computation
[1] Define problem and inputs.
[2] Secret sharing of inputs.
[3] Compute secret shares.
[4] Reconstruct the final result.
[5] Output verification.
[6] Decrypt the final result.

Substeps for: [2] Secret sharing of inputs.
[2.1] Generate random secret shares.
[2.2] Securely store secret shares.

Substeps for: [3] Compute secret shares.
[3.1] Encrypt secret share.
[3.2] Distribute encrypted shares.
[3.3] Compute ciphertext.
[3.4] Broadcast result.
```

For presentation purposes, we combine the first-layer steps and some of the substeps in one FSA and show it in Figure 5, where we indicate the first-layer steps and second-layer substeps in black and blue, respectively. This example demonstrates the capability of the proposed approach to automatically distill domain-specific knowledge into control-oriented automata representations from natural-language task descriptions, even when applied to highly specialized problems. We provide additional examples of domain-specific tasks in the Appendix.

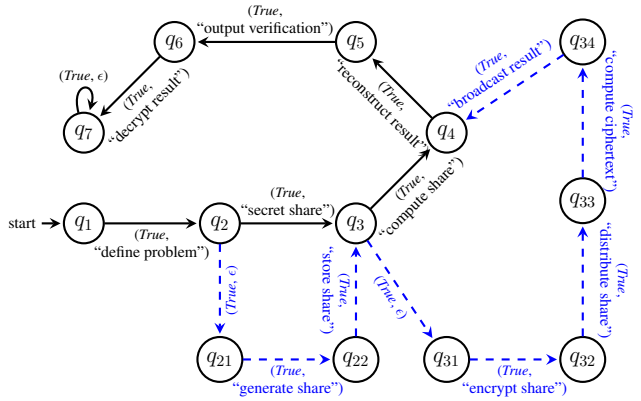


Figure 5: Example FSA for secure multi-party computation. The first-layer steps are represented by the states in black and the second-layer substeps are represented by the states whose transitions are in blue.

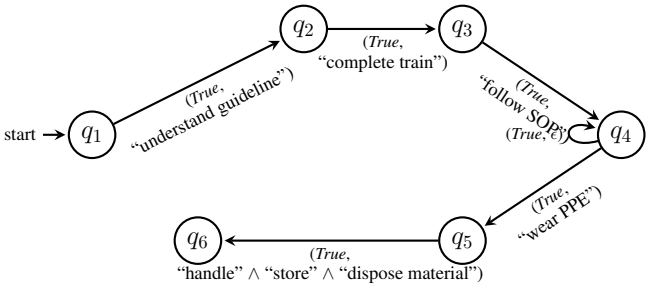


Figure 6: The distilled FSA-based controller for a biosafety laboratory test example.

Biosafety Laboratory: Distilling Domain-Specific Task Knowledge. Another example is the task of "passing a biosafety laboratory test". The biosafety laboratory (BSL) test examines if a technician is capable of doing biology or biomedical experiments. This is a specialized task in the biomedical field. We send a prompt to GPT-3 and get the following responses:

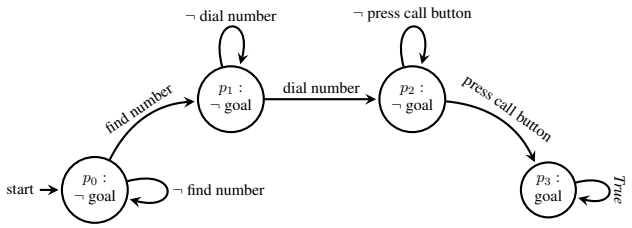
```
Steps for: passing a biosafety laboratory test
[1] Understand the biosafety guidelines.
[2] Complete necessary safety training.
[3] Follow the laboratory's SOPs (standard operating procedures).
[4] Wear appropriate PPE (personal protective equipment).
[5] Properly handle, store, and dispose of materials.
```

Then, we construct the FSA, displayed in Figure 6, to represent how to pass a BSL test.

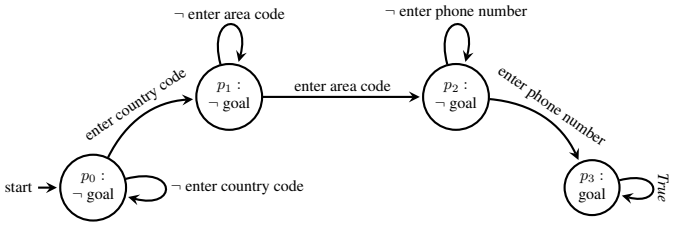
E Additional Examples of Controller Verification

Phone Call Example: Verifying Substeps. In this example, we build two models to separately verify the first-layer steps (black) and second-layer substeps (blue), of the controller for the phone call example, illustrated in Figure 4.

We build a first model (Figure 7a) to verify that the required first-layer steps are executed in the correct order. We then use the second model (7b) to check the correctness of the second-layer substeps represented in blue in Figure 4. For both models, the specifications will be satisfied if and only if all actions are taken in the correct consecutive order.



(a) The model used to verify the first-layer steps.



(b) The model used to verify the substeps for "dialing a number".

Figure 7: we use the models to verify the FSA for the task "making a phone call with a cellphone". For both models, we have a specification $\Phi = \Diamond\, goal$.

In both cases, the controller $\mathcal{C}$ from Figure 7 passes the verification step. This example demonstrates that first-layer and second-layer substeps (and more broadly, substeps at any layer of a task hierarchy) can be verified independently against separate models and specifications.