Automaton-Based Representations of Task Knowledge from Generative Language Models

Yunhao Yang YUNHAOYANG234@UTEXAS.EDU
Austin, TX, USA

Cyrus Neary CNEARY@UTEXAS.EDU
Austin, TX, USA

Ufuk Topcu UTOPCU@UTEXAS.EDU
Austin, TX, USA

Editors: G. Pappas, P. Ravikumar, S. A. Seshia

Abstract

Automaton-based representations 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, generated language models (GLMs) can automatically generate relevant text-based task knowledge. However, such text-based knowledge cannot be formally verified or used for sequential decision-making. We propose a novel algorithm named `GLM2FSA` that constructs a finite state automaton (FSA) encoding high-level task knowledge from a brief natural-language description of the task goal. `GLM2FSA` first sends queries to a GLM to extract task knowledge in textual form, and then it builds an FSA to represent this text-based knowledge. 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 specifications. We accordingly propose a procedure to iteratively refine the input queries to the GLM based on the outcomes, e.g., counter-examples, from verification. We apply the proposed algorithm to an autonomous driving system to demonstrate its capability for sequential decision-making and formal verification. Furthermore, quantitative results indicate the refinement method improves the probability of generated knowledge satisfying the specifications by 40 percent.

Keywords: Large Language Model, Automaton, Knowledge Representation, Formal Methods, Formal 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 of task knowledge enable users to formally verify the knowledge against externally provided specifications (Baier and Katoen, 2008). Hence, such representations are useful in many applications, such as lexical analysis of compilers (Brouwer et al., 1998; Arnaiz-González et al., 2018), reinforcement learning (Zhang et al., 2021; Valkanis et al., 2020; Xu et al., 2020), and program verification (Vardi and Wolper, 1986).

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 (Narendra and Thathachar, 1974). In general, these algorithms may require an excessive number of human queries, and it is often unclear how an automated oracle

© 2025 Y. Yang, C. Neary & U. Topcu.

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, which could be costly to obtain.

Although generative language models can help automatically distill high-level task knowledge, their textual outputs are not formally verifiable against external requirements. Existing GLMs are capable of generating realistic, human-like text in response to queries. Such text often encodes rich world knowledge. However, the textual outputs are not formally verifiable against user requirements, so they cannot be used directly in safety-critical applications.

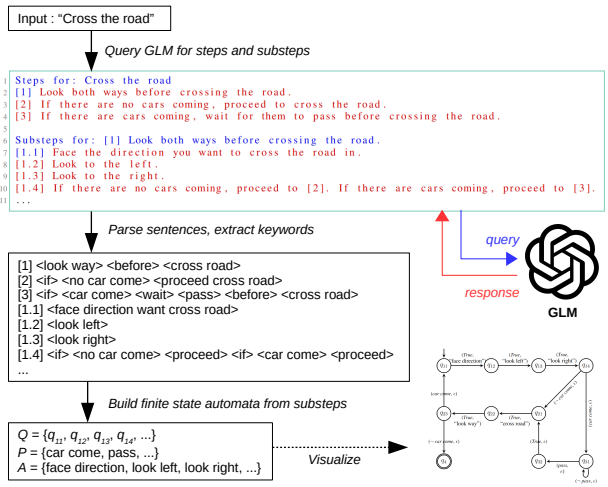


Figure 1: An illustration of the major steps in the GLM2FSA algorithm. A more detailed view of the output FSA for this example is presented in Figure 4.

We develop an algorithm named GLM2FSA to fill the gap between the outputs from GLMs 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 the task description to a GLM 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.

We accordingly propose a procedure to verify the controllers and to use the results of verification, e.g., counterexamples, as feedback to iteratively refine them through additional queries to the GLM. Such systematic verification-refinement procedure identifies and guards against potentially undesirable or nonsensical outputs from the language model, making it a necessary step towards the safe integration of GLMs into automated decision-making systems.

To the best of our knowledge, GLM2FSA is the first algorithm to construct automaton-based representations from textual knowledge extracted from GLMs. It is also the first algorithm to provide an approach to formally verify the knowledge from GLMs in the context of sequential decision-making and to use the results of the verification procedure to refine the extracted FSAs.

We demonstrate GLM2FSA's capabilities by applying the algorithm in an autonomous driving system. We show its ability to automatically distill task knowledge into control-oriented automaton-based representations and verify the automaton against system specifications. In doing so, we demonstrate that the algorithm yields controllers that have verifiable properties even though their task descriptions are provided in the form of natural-language sentences. Finally, in quantitative

experiments, we observe that the proposed refinement procedure improves the probability that generated controllers satisfy user requirements by 40 percent.

2. Related Work

Symbolic Knowledge Representations Many works focus on constructing symbolic representations of task knowledge or plans from natural language descriptions generated from GLMs (Vemprala et al., 2023; Ichter et al., 2022; Shah et al., 2022; Huang et al., 2022a,b). Several works extract information from text descriptions of given tasks and use that information to construct task-relevant knowledge graphs (West et al., 2022; Rezaei and Reformat, 2022; He et al., 2022; Lu et al., 2022; Thomason et al., 2020). Additionally, Vasileiou et al. (Vasileiou et al., 2022) and Wu et al. (Wu et al., 2022) both build logic-based or mathematical representations of the GLM's outputs, e.g., solutions to planning tasks. In contrast with the existing works, the automaton-based representations we produce are formally verifiable against user specifications and directly applicable in algorithms for sequential decision-making and reinforcement learning (Icarte et al., 2018; Xu et al., 2020; Neider et al., 2021; Neary et al., 2021; Fang et al., 2020).

Natural Language to Formal Language Existing works introduce approaches to transform natural language to formal language specifications (Vadera and Meziane, 1994; Baral et al., 2011; Sadoun et al., 2013; Ghosh et al., 2016; Fuggitti and Chakraborti, 2023). (Kate et al., 2005; Mora et al., 2024) induce transformation rules that map natural-language sentences into a formal query or command language. (Huang et al., 2022a,a) construct a form of actionable knowledge that machines can recognize and operate on. Existing works either cannot operate sequentially or cannot handle conditional transitions, e.g., multiple transitions from one state, which our work is capable of.

3. Preliminaries

Controller and Finite State Automaton A controller is a system component responsible for making decisions and taking actions based on the system's state. It can be mathematically represented as a mapping from the system's current state to an action that is executable in the task environment. In this work, a controller is a 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. Figure 4 depicts an example FSA.

We introduce a set P of atomic propositions 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 introduce another set P_A of atomic propositions for the output alphabets $A \coloneqq 2^{P_A}$. We allow a "no operation" symbol $\epsilon \in A$. We refer to the input and output alphabets as the *condition set* and *action set*.

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 (Kucera et al., 1967) for each token and builds *phrase structure* depending on 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 verb phrases (VP).

Definition 1 *A* Verb Phrase (VP) *is a tree-structured logical form headed by a verb.*

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 distinct.

4. Methodology

We propose an algorithm named `GLM2FSA`, which consists of two parts: GLM2Step and Step2FSA. GLM2Step uses a natural-language description of the task of interest to query the GLM and to obtain step-by-step task instructions in textual form. Step2FSA automatically constructs a controller from these text-based instructions.

Extracting Textual Knowledge First, we develop Algorithm 1 (Fig. 2) to distill task-relevant textual knowledge by iteratively prompting the GLM with structured natural-language queries. Given a task description of interest TASK_DESC, the algorithm returns a step-by-step textual description. We denote the step descriptions returned by the GLM after the first query as *first-layer* step descriptions. The substeps of the first-layer step are *second-layer* step descriptions. Similarly, the substeps of n^{th}-layer steps are $n + 1^{th}$-layer step descriptions. We denote the number of layers as *depth*.

Algorithm 1 depicts an iterative process that first queries the GLM for steps to accomplish the task description, and subsequently for substeps to accomplish these individual steps. This iterative process allows for the automated decomposition of the task description into a structured hierarchy of steps and substeps, up to a pre-specified depth.

Meanwhile, we define a set of *keywords* and set the GLM's bias to the keywords. Therefore, the GLM is intended to produce outputs containing the keywords. Then, we can use these keywords to define rules for building automata, as described in the following section.

Algorithm 1: Query the GLM for Task Instructions

```
procedure GLM2STEP(String TASK_DESC, integer DEPTH, List[String] keywords)  ▷ Obtain the instructions for a given task; depth is the number of layers of substeps.
    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
```

Algorithm 2: Natural Language to FSA

```
procedure STEP2FSA(keyword_handler, STEPS, keywords)  ▷ keyword_handler is a function, STEPS and keywords are lists of texts
    Q = [a state for each step]
    q_0 = Q[0], P, A, δ = {}, {ϵ}, {}
    for state_number in [0 : |Q| − 1] do
        i = step number of the current state
        VPs, Keys = parse(STEPS[I])
        if any(keywords) in Keys then
            keyword_handler(Q, VPs, Keys)  ▷ it add VPs to either Σ or A based on Table I.
        else
            δ(q_i, True, VPs, q_{i+1}) = 1, A := A∪VPs
        end if
    end for
    return P, A, Q, q_0, δ
end procedure
```

Figure 2: Algorithm 1 queries the GLM for task instructions. Algorithm 2 converts natural language to FSAs.

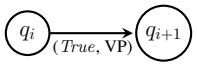
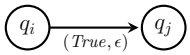
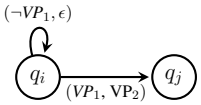
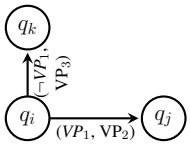
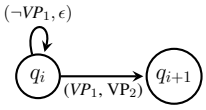
Category	Grammar	Transition Rule	Example
Default Transition	VP		[turn right]
Direct Transition	VP [j]		[proceed] [1]
Conditional Transition	**if** VP_1, VP_2 VP_2 **if** VP_1		[if] [no car], [cross]
Conditional Transition (if else)	**if** VP_1, VP_2. **if not** VP_1, VP_3. **if** VP_1, VP_2, **else** VP_3.		[if] [no car], [cross]. [if], [car] [stay].
Self Transition	**wait** VP_1 VP_2 VP_2 **after** VP_1 VP_1 **until** VP_2		[wait] [car pass] [cross road]

Table 1: Example transition rules defined for keywords under a specific natural language grammar. q_i is the state corresponding to the current step. In Conditional Transition and Self Transition, VP_1 is added to the condition set Σ, VP_2 and VP_3 are added to the action set A. In Default Transition, VP is added to the action set. In Direct Transition, we do not add VP to any set.

Building Automata from Textual Knowledge Algorithm 2 (Fig. 2) transforms step descriptions obtained from Algorithm 1 to FSAs. The algorithm first applies semantic parsing to classify the part-of-speech tags (Kucera et al., 1967) of each word in a sentence and builds word dependencies based on natural language grammar. Next, it extracts verb phrases and pre-defined keywords from the sentence. These keywords belong to a pre-defined set of words we use to define our grammar for automaton construction, such as *if* and *wait*. Sample keywords are highlighted as bold text in Table 1.

In Algorithm 2, **parse**$(\cdot)$ denotes the function that we implement to execute this keyword and verb phrase extraction process. In the implementation, we use the *spaCy* library for semantic parsing (Honnibal et al., 2020). The parse function takes a textual sentence and outputs a set of keywords KEYS and a set of verb phrases VPs. If a verb phrase VP includes one or more of the words *and*, *or*, *no*, or *not*, then the algorithm parses the VP as follows:

no/not VP_1 = $\neg$ VP_1,

VP_1 and VP_2 = $VP_1 \wedge$ VP_2,

VP_1 or VP_2 = $VP_1 \vee$ VP_2.

The algorithm constructs an FSA from the steps and the verb phrases within these steps. Recall that each step consists of a step number and a natural-language sentence. It transforms the verb phrases from each step into the components of an FSA: A finite set of states Q, a finite set of atomic propositions P, a finite action set A, a transition function δ, and an initial state q_0.

For each step, the algorithm adds a state q_i representing the current step i to Q. The algorithm uses **keyword_handler**$(\cdot)$ from Algorithm 2 (Fig. 2) to add transitions between states based on which grammar and keywords the step descriptions comply with. We provide examples in Table 1.

In particular, the **keyword_handler**$(\cdot)$ takes the set of states, extracted verb phrases, and keywords as inputs. It builds transitions and adds verb phrases to the action or condition sets, depending on the grammar. The **keyword_handler** is the key component that defines how we can translate texts into automata. We present more details in the caption of Table 1. Last, the algorithm defines the state corresponding to the first step as the initial state. It also adds a self-transition with input *True* and output "no operation" to the state corresponding to the final step.

Computation Complexity The algorithm processes each step at a time and builds a fixed set of states per step according to the **keyword_handler** (Table 1). Hence, the algorithm's complexity is $\mathcal{O}(N)$, where N is the total number of steps and substeps. Empirically, the algorithm's runtime is negligible compared to the GLM's latency.

5. Verification and Refinement

Given an automaton-based controller $\mathcal{C}$ output by `GLM2FSA`, we operate the controller in a system. We use a *model* $\mathcal{M}$—an abstract representation of the system—to verify the behavior of $\mathcal{C}$ against some task specifications of interest. If it fails to satisfy the task specification, we propose a procedure to refine the input prompt to the GLM and to update $\mathcal{C}$ accordingly.

Models of External Knowledge Mathematically, we define the *model* as a transition system $\mathcal{M} := \langle \Gamma_{\mathcal{M}}, Q_{\mathcal{M}}, \delta_{\mathcal{M}}, \lambda_{\mathcal{M}} \rangle$. $\Gamma_{\mathcal{M}} := 2^P$, where P is the set propositions from $\mathcal{C}$. $Q_{\mathcal{M}}$ is a finite set of states, $\delta_{\mathcal{M}} : Q_{\mathcal{M}} \times Q_{\mathcal{M}} \to \{0, 1\}$ 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 automatically check whether the controller, when implemented in the *model*, satisfies desired task specifications. Such systematic verification is necessary to identify and guard against undesirable or nonsensical outputs.

Task Specification We use linear temporal logic (LTL) (Pnueli, 1977) to define task specifications Φ that the controller $\mathcal{C}$ should satisfy, given the model $\mathcal{M}$. LTL is a formal language that expresses controller properties that evolve over time. It extends propositional logic by including temporal operators, such as $\Diamond$ ("eventually") and $\Box$ ("always"), which allow for reasoning about the controller's temporal behaviors. We define specifications Φ over atomic propositions in $P \cup P_A$.

Note: We do not expect the specification to contain full task knowledge, e.g., constraining all the behaviors in the task environment. Instead, the specification may only capture critical safety requirements. Therefore, we cannot directly synthesize a controller from the model.

Formal Verification To verify that the controller $\mathcal{C}$ satisfies the specification Φ given the model $\mathcal{M}$, we solve the following automated verification problem, $\mathcal{M} \otimes \mathcal{C} \models \Phi$, where $\mathcal{M} \otimes \mathcal{C}$ denotes the so-called *product automaton* describing the interactions of the controller $\mathcal{C}$ with the model $\mathcal{M}$. The formal definition of a product automaton is in Baier and Katoen (Baier and Katoen, 2008).

We leave the details of the automated verification problem to Baier and Katoen (Baier and Katoen, 2008). In this work, we use the NuSMV model checker (Cimatti et al., 2002) for this purpose. We provide the controller $\mathcal{C}$ and the model $\mathcal{M}$ to the model checker.

The outcome of the automated verification problem is binary: $\mathcal{C}$ either satisfies the specification Φ given the model, or it does not. However, if the controller fails to satisfy the specification, a counterexample is returned as a byproduct of the verification procedure. A counterexample is a sequence of states from the product automaton describing how the controller in the system fails Φ.

Vocabulary Alignment Due to the stochastic nature of generative models, the GLM may often output different phrases to represent the same concept. Furthermore, the verb phrases used to define the symbols of the externally provided model may use a different vocabulary than the one generated by the GLM. This mismatch in verb phrases could lead to multiple distinct verb phrases being used to describe conditions and actions that we intuitively understand as being the same. As a result, the automated verification procedure may yield unexpected failures because of its inability to recognize synonyms.

To remove these ambiguities, we automatically query the GLM to align the task knowledge to the model's symbols:

```
Rephrase the following steps with propositions {phrase 1, phrase 2,...}:
1. <step description>
2. <step description>
......
1. <aligned step description>
2. <aligned step description>
......
```

We build the controller $\mathcal{C}$ from the aligned step descriptions.

Refining the Automaton-Based Controllers If the controller $\mathcal{C}$ fails to satisfy the provided task specification Φ, we propose a counterexample-guided refinement procedure to iteratively refine $\mathcal{C}$ until it satisfies Φ.

The procedure asks the user to manually modify the input prompt to the GLM and to use the resulting outputs to update the controller. In detail, if the verification steps fail, the model checker generates a counterexample. The counterexample is a sequence of states $(p_1, q_1), (p_2, q_2), \ldots$ from the product automaton, where p_i is a state from $\mathcal{C}$ and q_i is a state from the model $\mathcal{M}$. The counterexample can automatically be converted into a sequence of the product automaton's output labels, as we call a *trajectory*, $\psi_0\psi_1, \ldots \in (2^{P \cup P_A})^*$ where $\psi_i \in \lambda_{\mathfrak{P}}((p_i, q_i), (p_{i+1}, q_{i+1}))$.

The user can then interpret this counterexample information and use it to modify the GLM's input prompt in a way that addresses the cause of the failure. After creating this modified input prompt, the user can use `GLM2FSA` to construct the updated controller. We provide detailed examples of this process, including illustrative counterexamples and the resulting user-specified refinements, in Section 6.

6. Experimental Results

We deploy the proposed algorithms through an autonomous driving system. We show the algorithms' capability to construct controllers operated in autonomous driving systems, verify the controllers against safety-critical specifications, and refine the controllers to meet those specifications.

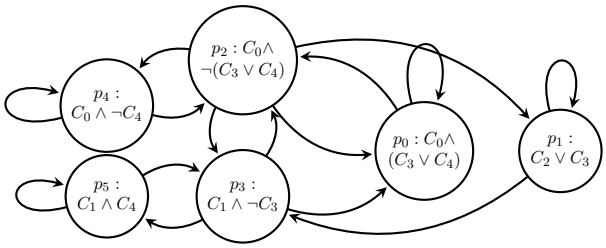


Figure 3: A model $\mathcal{M}_c$ for a driving system. For brevity, we present a sub-model for intersections with stop signs or traffic lights. In the figure, C_0 = stop sign observed, C_1 = green light observed, C_2 = red light observed, C_3 = pedestrian observed, C_4 = car observed. We annotate each node as "state: label."

Modeling the Autonomous Driving System Suppose the designer of the autonomous driving system provides an automaton-based model that captures all the possible behaviors of the system.

The model is over a set of atomic propositions $P = \{$stop sign observed, green light observed, red light observed, pedestrian observed, car observed $\}$. The autonomous driving system allows a set of actions $P_A = \{$ observe traffic, approach intersection, go straight, turn left, turn right, stop $\}$. The system designer also gives the two sets of propositions.

Driving Safety Specifications The system designer provides a set of system specifications to constrain the potential behaviors of any controllers operating in this system. These specifications are over the same propositions P and P_A. We include five sample specifications for our experiments:

$\Phi_1 = \square(\neg($ turn left $\vee$ turn right $\vee$ go straight $) \wedge$ pedestrian observed),

$\Phi_2 = \square($ stop sign observed $\rightarrow \Diamond$ stop $)$,

$\Phi_3 = \square($ red light observed $\rightarrow ($ stop $\vee$ turn right $))$,

$\Phi_4 = \square(\Diamond \neg($ pedestrian observed $\vee$ car observed$) \rightarrow \Diamond \neg$ stop $)$,

$\Phi_5 = \square($ green light observed $\rightarrow \Diamond \neg$ stop $)$.

Demonstrations of Hierarchical Controller Construction We construct several automaton-based controllers to demonstrate the capabilities of `GLM2FSA`. In particular, we provide examples to show that `GLM2FSA` can create FSAs that encode all of the knowledge required to describe the steps to complete a given task. In the following experimental case studies, we use the *GPT-4* as the GLM.

First, we apply the algorithm to construct an FSA for *first-layer* step descriptions. The queries and the responses from the GLM are as follows:

```
Steps for: Turn right at the stop sign
[1] Reach the stop sign intersection.
[2] Look for a safe gap to turn right.
[3] Turn right.
```

Next, we ask the algorithm to continue querying for the substeps of each step. We call these substep descriptions *second-layer* step descriptions.

```
...
Substeps for: [1] Reach the stop sign intersection.
[1.1] Approach the stop sign intersection.
[1.2] Stop at the intersection.

Substeps for: [2] Look for a safe gap to turn right.
[2.1] Look for oncoming traffic.
[2.2] If there are pedestrians, wait for the pedestrians.
[2.3] If there are coming vehicles, wait for the vechicles, else, proceed to [3].

Substeps for: [3] Turn right.
[3.1] Start turning right.
[3.2] Straighten the steering wheel as completing the turn.
```

Note that the number of layers is a design choice based on how detail the user expect the controller encodes.

Then, we query the GLM to align the vocabularies to the atomic propositions P and actions P_A provided from the autonomous system.

```
Rephrase the following steps with propositions {stop sign observed, car observed,
    ......}:
[1.1] Approach the stop sign intersection.
[1.2] ......

[1.1] Approach intersection. [1.2] Stop.
[2.1] Observe traffic.  [2.2] If pedestrian observed, wait for pedestrian.  [2.3] If
    car observed, wait for car, else, proceed to [3.1].
[3.1] Turn right. [3.2] Go straight.
```

We present the constructed FSA for the second-layer descriptions in Figure 4 (black and purple).

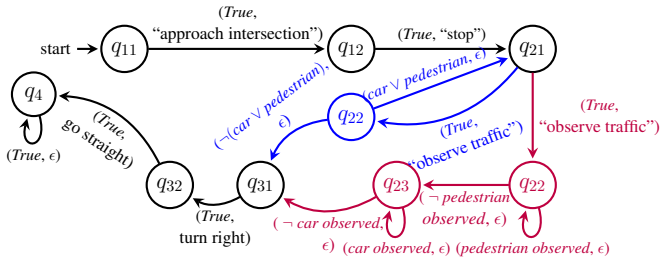


Figure 4: An FSA that represents all the substeps for turning left at the stop sign. States and transitions in purple and blue are before and after refinement, respectively.

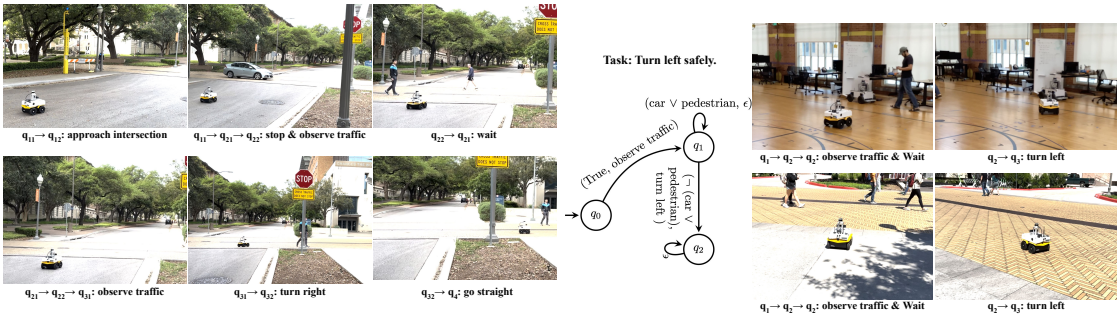


Figure 5: The left figures show a demonstration of grounding the constructed controller in Figure 4 on a real robot. The right figures present an example of a controller for "turn left safely."

Verification and Refinement We now consider verifying the controller in Figure 4, when implemented in the model in Figure 3, against the specification Φ_1. Φ_1 means "never turn left, right, or go straight if pedestrians are observed."

The verification step fails with a counterexample $(p_0, q_{11}), ..., (p_0, q_{31}), (p_0, q_{32})$. The counterexample indicates a scenario where the agent turns right when a pedestrian exists. This mistake could happen because the controller checks pedestrians and cars separately. When the controller ensures the nonexistence of cars, it directly turns right without checking for pedestrians again.

This is a potentially dangerous edge case that the GLM 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. To address the issue, the user modifies the input prompt:

```
Refine the steps to ensure the action "turn right" is performed when both pedestrian
    and car are clear.
......
[2.2] If pedestrian observed or car observed, proceed to [2.1], else, proceed to [3.1].
```

For brevity, we only show the modified steps. Then, we construct a new controller following the modified steps and present it in Figure 4 (black and blue). The refined controller passes all the verification steps, and hence it is finalized. Then, we ground this controller on real robots, where each proposition from the system model corresponds to an API in the robot's system. We present an example of successfully executing the controller in Figure 5.

We claim that if a controller satisfies all the specifications, the ground robot operating the controller also satisfies the specifications (with accurate perceptions).

Quantitative Evaluation We select autonomous driving as the application domain of our algorithm. Particularly, we query the GLM and construct controllers for eight tasks: {*turn left, turn right, go straight, make a U-turn*} *at the* {*traffic light, stop sign*}. We verify the controllers against the following specifications Φ_1,...,Φ_5 at the beginning of this section.

For each task, we query the GLM with different random seeds to construct 25 different controllers. We verify the total 200 controllers and get the probability of the generated controllers satisfying each specification. We additionally refine the controller through the counterexample-guided refinement procedure. Figure 6 presents the probabilities of the GLM-generated steps satisfying each specification after each refinement iteration.

Additionally, we compare the current state-of-the-art GLMs— GPT-4 (OpenAI, 2023), Llama 3 (Touvron et al., 2023), Qwen (Bai et al., 2023), and Claude 3.7 Sonnet (Anthropic, 2025) on their probabilities of generating specification-satisfied task knowledge, see Figure 6.

The results have shown the generalizability and real-world applicability of the `GLM2FSA` through various autonomous driving tasks. Overall, the constructed controllers for driving tasks achieve over 90 percent probability of satisfying the specifications within two refinement iterations.

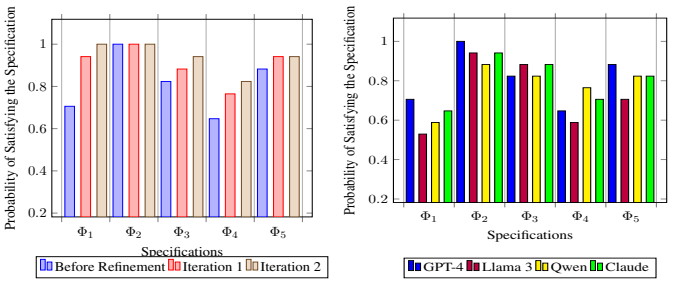


Figure 6: The left figure shows the probability of the GLM's generated task knowledge satisfying each specification before and after refinement. The right figure shows the probability of the generated task knowledge from different GLMs satisfying each specification.

7. Conclusions

Summary We provide a proof-of-concept for the automatic construction of automaton-based representations of abstract task knowledge from GLMs. We propose an algorithm, `GLM2FSA`, that accepts brief natural-language descriptions of tasks as input, queries a GLM, 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 a procedure to formally verify the constructed automata and to use the verification results to iteratively refine the inputs to the GLM.

Limitation and Future Direction We have focused on constructing controllers for high-level decision-making and verifying against high-level specifications. As a future direction, we can integrate perception models to collect more detailed information from the environment and use the information for low-level planning, e.g., trajectory planning in autonomous driving. Furthermore, we can explore the low-level planning problem under uncertainty raised by the perception models.

Acknowledgements

This research was supported by the Office of Naval Research (ONR) under Grant ONR N00014-24-1-2097 and the Army Research Laboratory under Grant ARL W911NF-24-1-0220.

References

Anthropic. Claude 3.7 sonnet. https://docs.anthropic.com/en/docs/about-claude/models/all-models, 2025.

Álvar Arnaiz-González, Jose-Francisco Díez-Pastor, Ismael Ramos-Pérez, and César García-Osorio. Seshat—a web-based educational resource for teaching the most common algorithms of lexical analysis. *Computer Applications in Engineering Education*, 26(6):2255–2265, 2018.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

Chitta Baral, Juraj Dzifcak, Marcos Alvarez Gonzalez, and Jiayu 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/.

Klaus Brouwer, Wolfgang Gellerich, and Erhard Plödereder. Myths and facts about the efficient implementation of finite automata and lexical analysis. In *Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 1–15, 1998. doi: 10.1007/BFb0026419. URL https://doi.org/10.1007/BFb0026419.

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, 2002. doi: 10.1007/3-540-45657-0_29. URL https://doi.org/10.1007/3-540-45657-0_29.

Xiaohan Fang, Jinkuan Wang, Chunhui Yin, Yinghua Han, and Qiang Zhao. Multiagent reinforcement learning with learning automata for microgrid energy management and decision optimization. In *Chinese Control and Decision Conference*, pages 779–784, 2020.

Francesco Fuggitti and Tathagata Chakraborti. Nl2ltl–a python package for converting natural language (nl) instructions to linear temporal logic (ltl) formulas. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 16428–16430, 2023.

Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried 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.

Mutian He, Tianqing Fang, Weiqi Wang, and Yangqiu Song. Acquiring and modelling abstract commonsense knowledge via conceptualization, 2022.

Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spacy: Industrial-strength natural language processing in python, 2020.

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, 2022a.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Tomas Jackson, Noah Brown, Linda Luu, Sergey Levine, Karol Hausman, and Brian 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, 2022b.

Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Anthony Valenzano, and Sheila A. McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. In Jennifer G. Dy and Andreas Krause, editors, *International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2112–2121, 2018. URL http://proceedings.mlr.press/v80/icarte18a.html.

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, 2022.

Rohit J. Kate, Yuk Wah Wong, and Raymond 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.

Henry Kucera, W. Nelson Francis, William Freeman Twaddell, Mary Lois Marckworth, Laura M. Bell, and John B. Carroll. Computational analysis of present-day american english. *International Journal of American Linguistics*, 35:71–75, 1967.

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

Federico Mora, Justin Wong, Haley Lepe, Sahil Bhatia, Karim Elmaaroufi, George Varghese, Joseph E Gonzalez, Elizabeth Polgreen, and Sanjit Seshia. Synthetic programming elicitation for text-to-code in very low-resource programming and formal languages. *Advances in Neural Information Processing Systems*, 37:105151–105170, 2024.

Kumpati 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.

Cyrus Neary, Zhe Xu, Bo Wu, and Ufuk Topcu. Reward machines for cooperative multi-agent reinforcement learning. In *International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '21, page 934–942, 2021.

Daniel Neider, Jean-Raphaël Gaglione, Ivan Gavran, Ufuk Topcu, Bo Wu, and Zhe Xu. Advice-guided reinforcement learning in a non-markovian environment. In *AAAI Conference on Artificial Intelligence*, pages 9073–9080, 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/17096.

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

Amir 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.

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

Driss Sadoun, Catherine Dubois, Yacine Ghamri-Doudane, and Brigitte 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.

Dhruv Shah, Blazej Osinski, Brian Ichter, and Sergey Levine. Lm-nav: Robotic navigation with large pre-trained models of language, vision, and action. In *Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pages 492–504, 2022.

Jesse Thomason, Aishwarya Padmakumar, Jivko Sinapov, Nick Walker, Yuqian Jiang, Harel Yedidsion, Justin W. Hart, Peter Stone, and Raymond J. Mooney. Jointly improving parsing and perception for natural language commands through human-robot dialog. *J. Artif. Intell. Res.*, 67: 327–374, 2020.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Sunil Vadera and F. Meziane. From english to formal specifications. *The Computer Journal*, 37: 753–763, 1994.

Anastasios Valkanis, Georgia A. Beletsioti, Petros Nicopolitidis, Georgios Papadimitriou, and Emmanouel A. Varvarigos. Reinforcement learning in traffic prediction of core optical networks using learning automata. In *International Conference on Communications, Computing, Cybersecurity*, 2020. doi: 10.1109/CCCI49893.2020.9256655. URL https://doi.org/10.1109/CCCI49893.2020.9256655.

Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Symposium on Logic in Computer Science*, pages 332–344, 1986.

Stylianos Loukas Vasileiou, William Yeoh, Tran Cao Son, Ashwin Kumar, Michael Cashmore, and Daniele Magazzeni. A logic-based explanation generation framework for classical and hybrid planning problems. *J. Artif. Intell. Res.*, 73:1473–1534, 2022.

Sai Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. ChatGPT for robotics: Design principles and model abilities, 2023. Published by Microsoft.

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, 2022. doi: 10.18653/v1/2022.naacl-main.341. URL https://doi.org/10.18653/v1/2022.naacl-main.341.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, New Orleans, LA, USA, 2022.

Zhe Xu, Ivan Gavran, Yousef Ahmad, Rupak Majumdar, Daniel Neider, Ufuk Topcu, and Bo Wu. Joint inference of reward machines and policies for reinforcement learning. In *International Conference on Automated Planning and Scheduling*, pages 590–598, 2020. URL https://ojs.aaai.org/index.php/ICAPS/article/view/6756.

Zhen Zhang, Dongqing Wang, and Junwei Gao. Learning automata-based multiagent reinforcement learning for optimization of cooperative tasks. *IEEE Transactions on Neural Networks and Learning Systems*, 32:4639–4652, 2021.

Appendix A. Additional Background and Definitions

Generative language model. A generative language model (GLM) produces human-like text *completion* from a given initial text (*prompt*). The produced texts continue filling the content from that prompt. Recent GLMs are deep-learning models with millions or billions of parameters; hence, they are called large-scale GLMs.

GPT-4 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: $\varphi \coloneqq p \in P_{\mathcal{M}} \mid \neg\varphi \mid \varphi \vee \varphi \mid \circ\varphi \mid \varphi \,\mathbf{U}\, \varphi$ 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. Recall that the controllers $\mathcal{C}$ output by GLM2FSA are defined as $\mathcal{C} \coloneqq \langle \Sigma, A, Q, q_0, \delta \rangle$ with input alphabet $\Sigma := 2^P$, output alphabet $A := 2^{P_A}$, and non-deterministic transition function $\delta : Q \times \Sigma \times A \times Q \to \{0, 1\}$.

We accordingly define the *product automaton* to be a transition system $\mathfrak{P} = \mathcal{M} \otimes \mathcal{C} \coloneqq \langle Q_{\mathfrak{P}}, \delta_{\mathfrak{P}}, q^{\mathfrak{P}}_{init}, \lambda_{\mathfrak{P}} \rangle$ with the following components:

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

Here, $\delta_{\mathfrak{P}} : Q_{\mathfrak{P}} \to 2^{Q_{\mathfrak{P}}}$ is a non-deterministic transition function, and $\lambda_{\mathfrak{P}} : Q_{\mathfrak{P}} \times Q_{\mathfrak{P}} \to 2^{P \cup P_A}$ is a labeling function on the transitions of the product automaton. 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. $\psi_0\psi_1, \ldots \in (2^{P \cup P_A})^*$ 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, we check that all possible labeled trajectories generated by the product automaton belong to the language defined by the LTL specification.

Appendix B. Additional Explanation to Table 1

Default Transitions. We define default transitions as transitions from the current state to the next state with a 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 $\delta(q_i, True, \text{VP}_i, q_{i+1})$ exists, unconditionally of the valuation of the atomic propositions in P (hence the corresponding transition's condition is $True$). A demonstration of the default transition is presented in the first row of Table 1.

Direct State Transitions. Next, we define a direct state transition, which is a transition from the current state q_i to a state other than the next state q_{i+1}. The direct state transition happens when there is a verb phrase in the step description that contains the number corresponding to another step. 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 also define a conditional transition, which is a transition that only happens when certain conditions are satisfied. During automaton construction, the algorithm will build a conditional transition when a step description contains the keyword *if*. The conditions themselves are defined as one or more atomic propositions in P.

The algorithm builds two transitions from each sentence according to the patterns illustrated in Table 1. The first transition consists of a starting state q_i, a conjunction of atomic propositions VP_1, a target state q_j, and a set of outputs VP_2. If the VP_2 does not lead to a direct state transition, then the transition will end at q_{i+1}. The second transition is a self-transition at q_i with a condition $\neg\text{VP}_1$ and with an output symbol ϵ (no action).

Self-Transitions. Finally, we define a self-transition, whose starting and target states are identical. The algorithm will construct self-transitions whenever a step description contains any of the keywords *wait*, *after*, or *until*. The self-transition will cause the automaton to stay in the current state until some logical condition is met, as specified by the keywords. We accordingly use the keywords and surrounding verb phrases to define a conditional transition that breaks the self-transition loop and proceeds to the next state.

Appendix C. Verification using NuSMV

Here we provide the NuSMV implementation used to verify the controllers from 4 under a model from 3.

C.1. Controller

```
1
2  MODULE environment ---------------------------------------------------------------
3
4  VAR
5      car_come: boolean;          -- cars are coming
6      car_pass: boolean;          -- all cars passed
7      turn_green: boolean;        -- the pedestrian traffic light turned green
8
9  FROZENVAR
10     traffic_light: boolean;     -- there is a pedestrian traffic light
11
12 MODULE actions ----------------------------------------------------------------
13
14 VAR
15     face_direction: boolean;    -- face direction of crossing
```

```
    look_left: boolean;          -- look left for cars
    look_right: boolean;         -- look right for cars
    look_way: boolean;           -- look both ways for cars
    cross: boolean;              -- cross the road

DEFINE COUNT_ACTIONS := count(
    face_direction,
    look_left,
    look_right,
    cross,
    look_way
);
DEFINE none := COUNT_ACTIONS = 0;   -- true iff no action is taken

INVAR COUNT_ACTIONS <= 1; -- cannot take two actions at once
INIT none;


MODULE controller_fig4(env,act) -------------------------------------------------

VAR state: {
    0,
    11,12,
    21,22,
    3
};
INIT            -- the initial state
    state=0;
DEFINE goal :=  -- the desired state
    state=3;

TRANS
    case
        state=0 & next(!env.traffic_light)
        : next(act.none & state=11);
        state=0 & next(env.traffic_light)
        : next(act.none & state=21);

        (state=11)
        : next(act.look_way & state=12);

        state=12 & next(env.car_come & !env.car_pass)
        : next(act.none & state=12);
        state=12 & next(!env.car_come | env.car_pass)
        : next(act.cross & state=3);

        (state=21)
        : next(act.look_way & state=22);

        state=22 & next(!env.turn_green)
        : next(act.none & state=12);
        state=22 & next(env.turn_green)
        : next(act.cross & state=3);

        goal
        : next(act.none & goal);
    esac;

MODULE controller_fig5(env,act) -------------------------------------------------

VAR state: {
    11,12,13,14,
    21,22,23,
    31,32,
    4
};
INIT            -- the initial state
    state=11;
DEFINE goal :=  -- the desired state
    state=4;

TRANS
    case
        state=11
        : next(act.face_direction & state=12);

        state=12
        : next(act.look_left & state=13);

        state=13
        : next(act.look_right & state=14);

        state=14 & next(!env.car_come)
        : next(act.none & state=21);
```

```
        state=14 & next(env.car_come)
        : next(act.none & state=31);

        state=21
        : next(act.cross & state=22);

        state=22
        : next(act.look_way & state=23);

        state=23 & next(!env.car_come)
        : next(act.none & state=4);
        state=23 & next(env.car_come)
        : next(act.none & state=11);

        state=31 & next(!env.car_pass)
        : next(act.none & state=31);
        state=31 & next(env.car_pass)
        : next(act.none & state=32);

        state=32
        : next(act.none & state=21);

        state=4
        : next(act.none & state=4);
    esac;


MODULE model(env,act) ---------------------------------------------------------

VAR state: {
    q_init,q_sink,q_goal
};
INIT            -- the initial state
    state=q_init;
DEFINE goal :=  -- the desired state
    state=q_goal;

TRANS
    case
        state=q_init & next(!act.cross)
        : next(state=q_init);
        state=q_init & next(act.cross & env.car_come)
        : next(state=q_sink);
        state=q_init & next(act.cross & !env.car_come)
        : next(state=q_goal);

        state=q_sink
        : next(state=q_sink);

        state=q_goal
        : next(state=q_goal);
    esac;

MODULE main -------------------------------------------------------------------

VAR env: environment;
VAR act: actions;

VAR controller: controller_fig4(env,act);
-- VAR controller: controller_fig5(env,act);
VAR model: model(env,act);

LTLSPEC NAME assume_guarantee :=
    (  TRUE -- ASSUMPTIONS

        -- the flow of cars is not continuous
        &  (G F !env.car_come)

        -- cars coming eventually passes
        &  (G (env.car_come -> F env.car_pass))

        -- if all cars passed,
        -- then none are coming towards the crossing
        &  (G (env.car_pass -> !env.car_come))

        -- when pedestrian light is green, no car can come on the crossing
        &  (G ((env.traffic_light & env.turn_green) -> (!env.car_come)))

    ) -> ( TRUE -- GUARANTEES

        -- the model terminal state is eventually reached
        &  (F model.goal)

```

```
        -- eventually, the controller rightfully thinks it achieved its goal
        &  (F (model.goal & controller.goal))

    );
```

C.2. Model for Driving Tasks

```
MODULE traffic_model
VAR
  Pedestrian_Observed : boolean;
  Car_Observed : boolean;
  Stop_Sign_Observed: boolean;
  Green_Light_Observed: boolean;
  Red_Light_Observed: boolean;
  Action : {Stop, Move_forward, Turn_left, Turn_right};

ASSIGN
  init(Car_Observed) := FALSE;
  next(Car_Observed) :=
    case
      TRUE: {TRUE, FALSE};
    esac;

  init(Pedestrian_Observed) := FALSE;
  next(Pedestrian_Observed) :=
    case
      TRUE : {TRUE, FALSE};
    esac;

  init(Stop_Sign_Observed) := FALSE;
  next(Stop_Sign_Observed) :=
    case
      TRUE: {TRUE, FALSE};
    esac;

  init(Green_Light_Observed) := FALSE;
  next(Green_Light_Observed) :=
    case
      TRUE: {TRUE, FALSE};
    esac;

  init(Green_Light_Observed) := FALSE;
  next(Green_Light_Observed) :=
    case
      TRUE: {TRUE, FALSE};
    esac;
```