LTLCodeGen: Code Generation of Syntactically Correct Temporal Logic for Robot Task Planning

Behrad Rabiei*   Mahesh Kumar A.R.*   Zhirui Dai   Surya L.S.R. Pilla   Qiyue Dong   Nikolay Atanasov

Abstract— This paper focuses on planning robot navigation tasks from natural language specifications. We develop a modular approach, where a large language model (LLM) translates the natural language instructions into a linear temporal logic (LTL) formula with propositions defined by object classes in a semantic occupancy map. The LTL formula and the semantic occupancy map are provided to a motion planning algorithm to generate a collision-free robot path that satisfies the natural language instructions. Our main contribution is LTLCodeGen, a method to translate natural language to syntactically correct LTL using code generation. We demonstrate the complete task planning method in real-world experiments involving human speech to provide navigation instructions to a mobile robot. We also thoroughly evaluate our approach in simulated and real-world experiments in comparison to end-to-end LLM task planning and state-of-the-art LLM-to-LTL translation methods.

Fig. 1: Our method is able to interpret natural language instructions for a robot navigation task, translate them into an LTL formula free of syntax errors, and plan a robot path that satisfies the instructions. The figure shows an example of a Jackal robot fetching an umbrella, bringing it to a backpack, and finally navigating to a microwave oven.

I. INTRODUCTION

The ability to understand and execute tasks specified in natural language is an important aspect of enabling autonomous robot assistants in our daily lives. A fundamental challenge is to translate high-level natural-language task descriptions into low-level, executable robot actions. Large Language Models (LLMs), such as GPT-4 [1], LLaMA-2 [2], and DeepSeek-R1 [3], have demonstrated remarkable natural language proficiency and language-based task reasoning [4], [5] and planning capabilities [6]. LLMs play different roles in the planning of robot task executions for natural language specifications [4], [5], [7]–[13]. An LLM can be a context extractor, which connects natural language concepts to real-world objects and locations, a process known as symbol grounding [14]. LLMs are also used as translators, for example, to convert natural language instructions into linear temporal logic (LTL) formulas [7], signal temporal logic (STL) [5], [13], planning domain definition language (PDDL) [13], or executable code [8], [10], [15]. An LLM can also operate as a task scheduler, which decomposes the task into sub-tasks, infers the order and the dependencies among the sub-tasks, and distributes them to different robots [5], [11]. LLMs also may cooperate with other multimodality models to better fuse environment feedback into task planning and execution [9], [12]. However, recent works [6], [16], [17] have shown that LLMs may not always

handle the whole process of converting natural language instructions into executable robot actions robustly without careful design of different components including symbol grounding, task decomposition, sub-task arrangement, formal logic translation, path generation, execution monitoring, and closed-loop feedback.

Many works have explored an end-to-end task planning approach where an LLM converts natural language instructions directly into robot actions [5], [8]–[10], [18], [19]. In this paper, we consider an alternative modular task planning approach where an LLM converts natural language instructions into a task model first and then composes it with a robot model and uses task and motion planning to generate robot actions. To ground the concepts in natural language instructions, our approach relies on a semantic occupancy map [20] constructed from range observations and semantic segmentation observations [21]. The map captures the free and occupied space as well as the categories of objects around the robot, allowing a connection to the objects that the natural language instructions may be referring to. We introduce a novel code generation approach, LTLCodeGen, to convert the natural language instructions into an LTL formula. The LTL formula and the semantic occupancy map are provided to a motion planning algorithm to generate a collision-free path that satisfies the task requirements. An example of our task planning approach is shown in Fig. 1.

The contributions of this paper are summarized as follows.

1) We propose LTLCodeGen, a method to translate natural language to syntactically correct linear temporal logic formulas using code generation.
2) We develop a modular task planning approach that converts natural language instructions to LTL over semantic occupancy map and uses motion planning to generate robot paths that satisfy the task requirements.
3) We compare LTLCodeGen with NL2LTL [7] and LLMs fine-tuned for LTL translation [22] on both human-written and synthetic datasets. We also compare our task planning approach with end-to-end LLM-based task planning on the GameTraversalBenchmark (GTB) [6]. Additionally, we conduct real-world robot experiments with tasks of different difficulty levels, and perform ablation studies to analyze the importance of different components in LTLCodeGen.

II. RELATED WORK

The components of natural language task planning include a) translating natural language instructions into a machine-interpretable format; b) constructing a scene representation with rich information to support instruction grounding and task planning; c) planning the task based on the translated instruction and the scene representation.

A. Natural Language Instruction Translation

Regardless of whether LLMs understand human language, their ability to transform text input to different text outputs based on specific requirements is evident [23]. Since LLMs are trained on internet-scale text data, the common knowledge stored in the model is helpful for inferring the human intent embedded in the natural language instructions. Converting instructions into machine-interpretable format involves two challenges. One is to associate concepts, e.g. objects, in the instructions, to physical locations in space, which is known as symbol grounding. Whether LLMs solve symbol grounding well remains unclear [23], [24] due to the debate on whether LLMs understand human language. The other challenge is to translate the symbol-grounded instruction to a machine-interpretable format that must correctly encode both the task objectives and constraints.

Recent works use different task representation formats, including LTL [7], [22], STL [25], [26], PDDL [13], code [8], [10], [11], [15], or others (e.g. waypoints [5], action scores [4], API calls [12]). Because LLMs are usually trained with a large corpus of code examples, they excel in writing high-quality code that calls predefined APIs based on the instruction, as shown in CaP [8], ProgPrompt [10], and Demo2Code [15]. However, it is still a challenge to generate code that provides an optimal and semantically correct plan for direct task execution. Other works use LTL [7] or STL [25], whose alignment with the user intent can be examined, to provide a task constraint for motion planning algorithms [27]–[29]. However, LLMs are not trained with various temporal logic examples and, even with carefully designed examples in the prompt, LLMs still generate syntactically incorrect LTL formulas occasionally. Some works [22], [26] propose to fine-tune LLMs for translating temporal logic. However, it is still difficult to scale up as the LTL formula becomes more complicated. To obtain both the syntax robustness of code generation and the post-processing convenience of temporal logic, we develop LTLCodeGen, which uses LLMs to generate code that outputs LTL formulas. LTL-CodeGen makes the whole system more robust to handle more complicated tasks and enables optimal planning for the instruction [7], [29].

B. Scene Representation

A semantically annotated scene representation is essential for accurate instruction interpretation. It must support symbol grounding, collision checking, and task progress feedback to ensure safe and correct execution. Some works aim to build such representations, such as a semantic occupancy octree [20], or a scene graph [30]–[34]. However, LLMs have a limited context window size, so the scene should be described to the LLM in a compact way. Existing works provide descriptions of involved objects only [8], [11], [25] or compress the scene in a concise format and then rely on the LLM to extract items related to the instruction [7]. Other works also consider obtaining scene information in different modalities [9], [35].

C. Task and Motion Planning

Prior works focus on different aspects of task and motion planning for natural language instruction. For improved feasibility, SayCan [4] fuses robot-state-based action affordances into the LLM planning procedure. InnerMonologue [18] continually injects new sensor observations as feedback into the LLM planner to correct the execution online.

Some works focus on multi-robot collaboration using a single LLM to reason about task decomposition and allocation [11] or a dialogue between multiple LLMs [5]. In contrast, our work focuses on generating an optimal path satisfying the LTL formula that correctly reflects the instruction.

III. PROBLEM STATEMENT

Enabling mobile robots to navigate following human instructions in natural language is essential for effective human-robot interaction. We consider a mobile robot with state $x_t \in \mathcal{X} \subseteq \mathbb{R}^n$ at time t equipped with an RGBD sensor. The robot first needs to construct a map of its environment, which stores information needed to understand the semantics of a human instruction. To ensure safe navigation, the map should also capture the presence of obstacles correctly.

Definition 1. A *semantic occupancy map* is a function $m : \mathcal{X} \to \mathcal{C}$, which associates a robot state $\mathbf{x} \in \mathcal{X}$ to a semantic category $m(\mathbf{x}) \in \mathcal{C}$ present at state $\mathbf{x}$. The set $\mathcal{C}$ is a set of semantic categories, including object classes, a FREE class that represents unoccupied free space, a NULL class that represents occupied space with unknown semantic category, and an UNKNOWN class for unobserved space.

Fig. 2 shows an example of a semantic occupancy map. For example, given a semantic occupancy map of a house, the

robot may be asked to tidy up the dining table and place the dishes in a dishwasher. Using the map, the robot can connect the instructions to the physical locations of the table and dishwasher and plan its motion to navigate to these locations safely and follow the sequence required by the instructions.

To construct a connection between the natural language instructions and the semantic occupancy map, we define boolean propositions that capture the task requirements.

Definition 2. An *atomic proposition* is a boolean function $p_c(\mathbf{x})$ that evaluates true when the robot state $\mathbf{x}$ is sufficiently close to a class $c \in \mathcal{C}$ in a semantic occupancy map m:

$$p_c(\mathbf{x}) = \begin{cases} \text{true}, & \exists \mathbf{y} \in \mathcal{X}, m(\mathbf{y}) = c, \|\mathbf{x} - \mathbf{y}\| \leq r_c \\ \text{false}, & \text{otherwise}, \end{cases} \quad (1)$$

where r_c is a distance threshold associated with class c. Denote the set of atomic propositions by $\mathcal{AP} = \{p_c | c \in \mathcal{C}\}$.

To capture which atomic propositions are satisfied at different robot states, we define a label map.

Definition 3. Given a semantic occupancy map m and atomic propositions $\mathcal{AP}$, a *label map* $l : \mathcal{X} \to 2^{\mathcal{AP}}$ associates a robot state $\mathbf{x}$ to the set of atomic propositions that evaluate true at $\mathbf{x}$, i.e., $p_c(\mathbf{x}) = \text{true}$ for all $p_c \in l(\mathbf{x}) \subseteq \mathcal{AP}$.

The atomic propositions satisfied along a path $\mathbf{x}_{1:T} := \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T$ are obtained from the label map as $l(\mathbf{x}_{1:T}) := l(\mathbf{x}_1), l(\mathbf{x}_2), \ldots l(\mathbf{x}_T)$. The sequence $l(\mathbf{x}_{1:T})$ is called a *word* and it encodes the task requirements satisfied along the robot path $\mathbf{x}_{1:T}$. Finally, we use the notation $l(\mathbf{x}_{1:T}) \models \mu$ to indicate that all requirements of a task μ are satisfied by a word $l(\mathbf{x}_{1:T})$. We define the motion planning problem associated with a natural language task μ as follows.

Problem. Given a semantic occupancy map m, a natural language navigation task μ defined in terms of semantic classes $\mathcal{C}$, a cost function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_{>0}$, and an initial robot state $\mathbf{x}_1 \in \mathcal{X}$, plan a path $\mathbf{x}_{1:T}$ that satisfies μ with minimum cost:

$$\min_{T \in \mathbb{N}, \mathbf{x}_{1:T}} \sum_{t=1}^{T-1} d(\mathbf{x}_t, \mathbf{x}_{t+1}) \quad (2)$$

$$\text{s.t.} \quad l(\mathbf{x}_{1:T}) \models \mu; \quad m(\mathbf{x}_t) = \text{FREE}, \quad t = 1, \ldots, T.$$

A key aspect of solving the problem above is to ground the task specification μ to states in the map m with associated semantic classes and atomic propositions and to encode the dependencies among the atomic propositions according to μ. In the next section, we propose an LLM code generation technique to generate LTL from μ using the atomic propositions defined by the semantic occupancy map m.

IV. ROBOT TASK PLANNING VIA LTLCODEGEN

Our approach to the task planning problem described in the previous section begins with constructing a semantic occupancy map, which we discuss in Sec. IV-A. Next, in Sec. IV-B, we introduce our main contribution, LTLCode-Gen, a method to translate natural language to syntactically correct LTL formulas over semantic occupancy map atomic

Fig. 2: Semantic occupancy map of an office environment. Each voxel encodes both occupancy and semantic classification, represented by distinct colors: cyan indicates walls, orange and blue denote office tables, sky blue represents a cart, and lime green corresponds to carpeted floors.

propositions using code generation. Finally, in Sec. IV-C, we present a task planning approach that takes the semantic occupancy map and the LTL formula and generates a collision-free robot path that satisfies the task requirements. Fig. 3 shows an overview.

A. Semantic Occupancy Mapping

Before the robot can interpret a desired task, it needs a representation of the environment that captures occupancy (for collision avoidance) and semantic classes (for task grounding). We assume that the robot is able to explore the environment first (either autonomously or through teleoperation) to construct a semantic occupancy map (Definition 1) using an RGB and range sensor. Our approach requires three components: an odometry algorithm (e.g., Direct LiDAR Odometry (DLO) [36]) that estimates the robot's pose, a semantic segmentation algorithm (e.g., YOLO [21]) that assigns semantic labels from $\mathcal{C}$ to image pixels in real time, and a semantic occupancy mapping algorithm (e.g., SSMI [20]) to fuse semantically labeled point clouds, constructed from the segmentation and range measurements, into a semantic occupancy map. We chose DLO [36] for localization, YOLO [21] for semantic segmentation, and SSMI [20] for semantic occupancy mapping due to their efficiency, robustness, and the availability of open-source code. Other alternatives would also be compatible with our approach.

While SSMI [20] constructs a 3D semantic occupancy map, in our experiments we consider a ground wheeled robot navigating on a 2D plane. We project the 3D map m_{3D} onto this plane and discard voxels outside of a vertical region of interest $\mathcal{Z} = \{z | z_{\text{ground}} < z < z_{\text{ceiling}}\}$. Specifically, we obtain a 2D semantic occupancy map m as follows: 1) $m(x, y) = \text{UNKNOWN}$ if the entire vertical column of voxels at (x, y) is unobserved; 2) $m(x, y) = \text{FREE}$ if the vertical column of voxels at (x, y) contains observed voxels and all of them are free; 3) $m(x, y) = \text{NULL}$ if all the occupied voxels in the column at (x, y) are labeled with NULL; 4) otherwise, we pick the semantic class from $\bigcup_{z \in \mathcal{Z}} \{m_{3D}(x, y, z)\} \setminus \{\text{UNKNOWN}, \text{FREE}, \text{NULL}\}$ with the largest z location. The robot uses $m(\mathbf{x})$ to interpret the

Fig. 3: Overview of robot task planning using LTL code generation. Given a semantic occupancy map (left), we convert natural language instructions into a syntactically correct LTL formula using an LLM code generation approach (top). A label map of atomic propositions (bottom, middle), obtained from the map semantic information, and an Büchi automaton (top, right), obtained from the LTL formula, are used as inputs to a motion planning algorithm to generate a collision-free semantically valid robot path that executes the task.

TABLE I: Grammar for LTL formulas ϕ and φ.

p_c	(Atomic Proposition)	$\phi \vee \varphi$	(Or)	$\phi \mathbf{U} \varphi$	(Until)
$\neg \phi$	(Negation)	$\phi \Rightarrow \varphi$	(Imply)	$\mathbf{F} \phi$	(Eventually)
$\phi \wedge \varphi$	(And)	$\mathbf{X} \phi$	(Next)	$\mathbf{G} \phi$	(Always)

natural language instructions by identifying regions that contain categories of interest and to ensure collision-free task planning and execution.

B. Translating Natural Language to LTL

This section describes our main contribution, an approach to translate natural language instructions to LTL using code generation, ensuring syntactic correctness of the resulting formulas. LTL is a widely used and sufficiently expressive formalism for expressing a variety of robot tasks [37].

LTL formulas are obtained from a set of atomic propositions, logic operators $(\wedge, \vee, \neg)$, and temporal operators $(\mathbf{U}, \mathbf{F}, \mathbf{X}, \mathbf{G}, \Rightarrow)$ with syntax summarized in Table I. We assume that the task μ can be translated to a syntactically co-safe (sc) LTL formula [38]. Any word that satisfies an sc-LTL formula consists of a finite satisfying prefix followed by any infinite continuation that does not affect the formula's truth value. Thus, sc-LTL formulas allow task satisfaction verification and task planning over finite words.

Although LLMs can translate natural language to LTL formulas using several examples provided with the prompt, LTL formulas generated this way are prone to syntax errors, such as missing parentheses or operands. Our key idea is to use an LLM to generate (Python) code that describes a natural language instruction μ using a predefined library of functions, defining the LTL logic and temporal operators. The generated code is then executed to output an LTL for-

mula ϕ_μ. Our LTLCodeGen approach inherits the robustness and expressiveness capabilities of LLM code generation and the convenience to verify the LTL syntax as a consequence of the code syntax correctness.

```
IV.1: Object Identification

Your task is to convert the object names to their unique
id based on given object id correspondences. Use the
object IDs provided in the object ID correspondences for
conversion. The conversion should take the context of
each sentence into account, so that objects can be
correctly correlated to the text.
Here are a few examples:
Object ID correspondence:
    'object_28' : 'refrigerator'
    'object_31' : 'bottle'
    ...
Input text: Take the teddy bear, then pick the bottle.
Always avoid the refrigerator.
Output text: Take object_36, then pick object_31. Always
avoid the object_28.
...
Using the provided examples, convert the objects in the
following text into their unique IDs.
Object ID correspondence: {object ids}
Input text: {natural language instruction}
Output text:
```

Object Category Identification: To ground the semantics of the natural language instruction μ with respect to the categories in the semantic occupancy map $m(\mathbf{x})$, LTLCodeGen first rephrases the instruction μ with unique IDs of semantic classes in $\mathcal{C}$. As shown in Code IV.1, we first describe the task of replacing the objects in the instruction with unique IDs. Then, we provide examples of inputs and corresponding expected outputs to the LLM. Finally, we provide the unique IDs of semantic classes in $\mathcal{C}$, then ask for the instruction $\mu_\mathcal{C}$ rephrased with IDs, from which we identify categories $\mathcal{C}_\mu$

present in both the map and the instruction.

Code Generation: We first briefly describe the code writing task in comments. Then, to make sure that the LLM generates code without undefined variables or functions, we show the available variables and library functions via import statements. As shown in Code IV.2, we first import the function `ap(obj)` for creating atomic propositions, and then the predefined LTL operators, such as `ltl_and(a, b)`. Code IV.3 shows the implementation of `ltl_and(a, b)`. Other LTL operator functions are defined similarly. The global variable `prefix` is used to control the LTL format. It is not necessary to provide the implementations of the library functions to the LLM, which wastes the limited number of input tokens. We hide the implementation details of these LTL operator functions and, instead, show examples of how to use them as in Code IV.4.

IV.2: Actions and Predefined Functions

```python
# Please help write code to translate the instruction
into an LTL formula.
# Necessary functions and variables are imported.
from ltl_operators import ap  # ap(obj)
from ltl_operators import ltl_and, ltl_or, ltl_not,
ltl_until, ltl_eventually, ltl_always, ltl_imply
```

IV.3: Implementation of `ltl_and`

```python
prefix = True  # The generated LTL is in prefix format
def ltl_and(a: str, b: str):
    if prefix:
        return f"& {a} {b}"
    return f"({a}) & ({b})"
```

IV.4: Code Example

```python
def example_1():
    """
    Reach object_2 and object_1
    """
    # explanation of the instruction:
    # object_1 and object_2 are both reached at some
    point in the future, but no specific order is
    mentioned explicitly or implicitly.
    # create atomic propositions for objects
    reach_obj_2 = ap("object_2")  # reach(object_2)
    reach_obj_1 = ap("object_1")  # reach(object_1)
    # describe the constraints in the instruction
    c1 = ltl_eventually(reach_obj_2) # Reach object_2...
    c2 = ltl_eventually(reach_obj_1) # Reach object_1...
    c3 = ltl_and(c1, c2)  # Reach object_2 and object_1
    return c3
```

The code examples not only show the basic usage of the predefined functions but also demonstrate how to write the code based on the rephrased instruction $\mu_\mathcal{C}$. The LLM should first explain the instruction in comments, then create atomic propositions for objects involved in the instruction, next describe the instruction with the LTL operator functions, and finally return the LTL formula.

As shown in Code IV.5, we describe the code writing task again with the requirements for the output format, i.e., the LLM output should only contain the complete implementation of the `question` function for the instruction $\mu_\mathcal{C}$.

IV.5: Provide the Instruction to Translate

```python
# Now, please finish the following Python code for
translating the instruction to LTL formula.
# The returned output should only contain the code that
starts with `def` and ends with `return` statement.
def question():
    """
    {instruction}{previous_answer}{failure_reason}
    """
```

Code Syntax Checks: The LLM may generate code that contains syntax errors, such as undefined functions, too many arguments for a function call, etc. To ensure that the generated code is syntactically correct, we execute it and provide the error message from any syntax errors back to the LLM. If an error is detected, we fill the `previous_answer` and `failure_reason` fields in Code IV.5 with the latest generated code and a short description of the error, respectively. In our practice, GPT-4o [1] rarely generates Python code with syntax errors, which are easy to fix in a second query to the LLM. When the code is free of syntax errors, it produces an LTL formula ϕ_μ free of LTL syntax errors.

C. Planning

Given the atomic propositions $\mathcal{AP}_\mu$ related to the instruction μ and the LTL formula ϕ_μ, synthesized by LTL-CodeGen, we can convert ϕ_μ into a Büchi automaton via a translation tool, such as Spot [39]. Büchi automata are more expressive than LTL formulas, and a Büchi automaton that recognizes the same language as an LTL formula can always be constructed [40].

Definition 4. A deterministic Büchi automaton is a tuple $\mathcal{B} = (\mathcal{Q}, \Sigma, T, \mathcal{F}, q_1)$, where $\mathcal{Q}$ is a finite set of states, Σ is a finite set of inputs, $T : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is a transition function that specifies the next state $T(q, \sigma)$ from state $q \in \mathcal{Q}$ and input $\sigma \in \Sigma$, $\mathcal{F} \subseteq \mathcal{Q}$ is a set of final (accepting) states, and $q_1 \in \mathcal{Q}$ is an initial state.

Then, to plan a collision-free robot path that satisfies the LTL formula ϕ_μ obtained from LTLCodeGen, we first construct a product planning graph $\mathcal{G}$ based on the (semantic) occupancy map $m(\mathbf{x})$, the label map $l(\mathbf{x})$, and the automaton $\mathcal{B}_{\phi_\mu}$. With eight-direction movements (cardinal and diagonal) in the (semantic) occupancy map $m(\mathbf{x})$, we construct the graph as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \mathcal{X} \times \mathcal{Q}$ is the set of nodes, $\mathcal{E}$ is the set of transitions from $\mathbf{v}_i = (\mathbf{x}_i, q_i)$ to $\mathbf{v}_j = (\mathbf{x}_j, q_j)$ with a cost $d(\mathbf{x}_i, \mathbf{x}_j)$. Such a transition exists when 1) $\mathbf{x}_j$ can be reached from $\mathbf{x}_i$ by a single-step movement in the map $m(\mathbf{x})$; 2) both $\mathbf{x}_i$ and $\mathbf{x}_j$ are collision-free: $m(\lambda\mathbf{x}_i + (1-\lambda)\mathbf{x}_j) = \text{FREE}, \forall\lambda \in [-r_o/\alpha, 1+r_o/\alpha]$, where $\alpha = \|\mathbf{x}_i - \mathbf{x}_j\|_2$, and r_o is a safe margin; 3) the automaton transitions are respected: $q_j = T(q_i, l(\mathbf{x}_i))$.

Finally, we apply the A* algorithm [41] to search the graph $\mathcal{G}$ with a consistent LTL heuristic [7], defined as:

$$h(\mathbf{x}, q) = \min_{\mathbf{y} \in \mathcal{X}} \left[d(\mathbf{x}, \mathbf{y}) + g(l(\mathbf{y}), T(q, l(\mathbf{y}))) \right],$$
$$h(\mathbf{x}, q) = 0, \quad \forall q \in \mathcal{F}, \tag{3}$$

TABLE II: LTL translation accuracy (%). Results from the BART baselines are from the original paper [22]. BART-FT-RAW-human is shown as BART-human in the table, and BART-FT-RAW-synthetic is shown as BART-syn.

	Drone		Cleanup		Pick	
	human	syn.	human	syn.	human	syn.
LTLCodeGen	**99.87**	**99.94**	**99.05**	**98.99**	**99.19**	**98.92**
NL2LTL [7]	98.66	98.30	95.36	95.21	97.31	97.18
BART-human	90.78	N/A	97.84	N/A	95.97	N/A
BART-syn. [22]	69.39	N/A	78.00	N/A	81.45	N/A

where the function $g : 2^{\mathcal{AP}} \times \mathcal{Q} \mapsto \mathbb{R}_{\geq 0}$ is defined as:

$$c_l(l_1, l_2) = \min_{\mathbf{x}_1, \mathbf{x}_2 : l(\mathbf{x}_1)=l_1, l(\mathbf{x}_2)=l_2} c(\mathbf{x}_1, \mathbf{x}_2), \quad (4)$$

$$g(l, q) = \min_{l' \in 2^{\mathcal{AP}}} c_l(l, l') + g(l', T(q, l')), \quad (5)$$

which can be pre-computed via dynamic programming on $\mathcal{B}_{\phi_\mu}$. The search terminates when the accepting state in $\mathcal{F}$ is reached, meaning the path satisfies the LTL formula ϕ_μ and, thus, the corresponding natural language instruction μ. If an accepting path $\mathbf{x}_{1:T}$ is found, a tracking controller is then used to drive the robot along the path.

V. EXPERIMENTS

We conduct a series of comprehensive experiments to evaluate LTLCodeGen and our complete task planning method and compare the performance against multiple baselines. First, we compare LTLCodeGen to the LLM-based NL2LTL [7] and two fine-tuned LLM models [22] on three datasets (Drone, Cleanup, and Pick) to gauge translation accuracy from natural language to LTL. Next, we demonstrate our task planning approach in both simulation and real-world environments, where tasks of varying complexity are tested. We also compare our method to an end-to-end LLM-based task planner on the GameTraversalBenchmark (GTB) [6]. Finally, we conduct an ablation study to investigate how different LLMs, as well as the inclusion of explanations and comments, affect the performance of LTLCodeGen.

A. Evaluation of LTLCodeGen

Using GPT-4o [1], we evaluate our LTLCodeGen and three baselines, NL2LTL [7], BART-FT-RAW-human and BART-FT-RAW-synthetic [22] on three datasets (Drone, Cleanup, and Pick) from the fine-tuned LLM [22] paper. The three datasets provide both human-written and LLM-augmented synthesized natural language instructions. For each dataset, we adjust the prompt for LTLCodeGen and NL2LTL based on a few examples randomly drawn from the datasets. The prompt for each dataset is given only 20 (Drone), 12 (Cleanup) and 5 (Pick) examples. Our method and NL2LTL are both tested with all human-written and synthetic instructions. In contrast, BART-FT-RAW-human is evaluated by 5-split cross-validation on human-written data, and BART-FT-RAW-synthetic is trained with synthetic data and tested with human-written data. As shown in Table II, LTLCodeGen outperforms the baselines significantly on both human-written and synthesized instructions, which indicates that LTLCodeGen generalizes well and generates the correct LTL formulas robustly. NL2LTL also performs better than

Fig. 4: Example path generated by our method for a GTB map. The robot is asked to *obtain the heartstone*, *rally the creatures of Verdanthorn*, then *defeat Drakon's Lieutenants*, and finally *defeat Drakon and restore peace to Verdanthorn*. Note that bush tiles are considered walkable in this map.

TABLE III: Evaluation of our task planning method on the GameTraversalBenchmark (GTB) [6]. Higher Accuracy and lower MPL indicate better performance. Both methods are evaluated using GPT-4o [1]. Results for GTB [6] are gathered by using GPT-4o to plan the path directly via prompting as described in their work.

Method	Accuracy (%) ↑	MPL ↓
GTB [6]	7.84	85.42
Ours	**78.91**	**82.49**

the fine-tuned LLMs but worse than LTLCodeGen because LLMs are not as good at writing LTL formulas. In our experiments, NL2LTL fails mostly due to the LTL syntax errors that LLMs cannot fix within three retries or due to being semantically mismatched with the instructions.

B. Evaluation on GameTraversalBenchmark (GTB)

We evaluate our complete task planning approach on the GameTraversalBenchmark (GTB) [6]. GTB is an LLM-generated dataset that provides 150 different 2D maps with various objects, game stories, and task objectives. For each experiment, we convert the map into the 2D semantic occupancy map format and generate an instruction that combines all the task objectives. For example, if the objectives are *objective₁*, *objective₂*, and *objective₃* the instruction provided to the LLM would be: Complete *objective₁*, then complete *objective₂*, and then *objective₃*. Our system consumes the map and the instruction to generate a path that satisfies all the task objectives. We report two metrics in Table III:

- **Accuracy (↑):** the percentage of tests where the agent successfully reaches the exact objective coordinates;
- **Mean Path Length (MPL) (↓):** the average length of paths taken by the agent to complete each task.

The results in Fig. 4 and Table III show that our system outperforms the end-to-end GPT-4o-based planner from GTB [6]. However, there are cases where our planner is unable to generate a path due to the absence of a valid path satisfying the LTL in the environment. For example, achieving *objective₁* may require passing through a later objective,

TABLE IV: LTL generation comparison: LTLCodeGen vs NL2LTL [7]. Two LLMs, GPT-4o and GPT-4o-mini, are tested.

Method	Success (%)↑				Semantic Failure (%)↓				Syntactic Failure (%)↓				Runtime (s)↑
Tier	1	2	3	4	1	2	3	4	1	2	3	4	Total
LTLCodeGen-4o	**100**	**100**	**100**	**70.0**	**0**	**0**	**0**	**30.0**	**0**	**0**	**0**	**0**	7.07 ± 4.31
NL2LTL-4o [7]	100	90.0	46.7	60.0	0	10.0	40.0	40.0	0	0	13.3	0	**2.01 ± 1.12**
LTLCodeGen-4o-NoExplanation	100	100	60.0	50.0	0	0	40.0	50.0	0	0	0	0	5.55 ± 3.20
LTLCodeGen-4o-NoComment	100	100	93.3	70	0	0	6.7	30.0	0	0	0	0	6.06 ± 3.06
LTLCodeGen-4o-mini	90.0	70.0	26.7	0	10.0	30.0	73.3	100	0	0	0	0	4.90 ± 3.40
NL2LTL-4o-mini	80.0	70.0	13.3	0	20.0	10.0	26.7	80.0	0	20.0	60.0	20.0	2.15 ± 0.93
LTLCodeGen-4o-mini-NoExplanation	100	80.0	20.0	0	0	20.0	80.0	100.0	0	0	0	0	4.49 ± 2.22
LTLCodeGen-4o-mini-NoComment	90.0	80.0	40.0	0	10.0	20.0	60.0	100.0	0	0	0	0	3.77 ± 1.25

such as *objective₃*, which conflicts with the generated LTL.

C. Real-World Experiments

We evaluate our system in a mock indoor environment containing eight categories: personal items (backpack, laptop, umbrella), common household objects (TV, potted plant, microwave, chairs), and a person for a realistic setting.

The task design includes four tiers of natural language tasks progressively increasing in difficulty:

- *Tier 1: Single-object tasks.* Straightforward instructions referencing one object (e.g., "Visit the chair").
- *Tier 2: Multi-object tasks with avoidance.* Missions involving one or more objects while *avoiding* others (e.g., "Go to the table and avoid chairs") or *applying a condition* (e.g., "Go to the backpack if you are near a laptop").
- *Tier 3: Multi-object tasks with temporal constraints.* Complex missions requiring multiple visits or precise ordering (e.g., "Visit the desk, then the couch, and do not go by the chair").
- *Tier 4: Ambiguous, context-based tasks.* Instructions relying on inference and context rather than explicit object naming (e.g., "It's valentine's day and I have no date. Let's watch a romcom.").

The evaluation includes 10 scenarios for Tier 1, 10 for Tier 2, 15 for Tier 3, and 10 for Tier 4, totaling 45 scenarios. Our planner runs using LTL specifications generated by LTL-CodeGen or NL2LTL [7], both with GPT-4o and GPT-4o-mini [1]. The robot's trajectory undergoes visual inspection to determine task success and the following metrics:

- **Success Rate**: Fraction of tasks the robot completes.
- **Semantic Error Rate**: Fraction of tasks where the plan does not match user intent.
- **Syntactic Error Rate**: Fraction of tasks where the LLM generates invalid LTL after three retries.
- **LLM Runtime**: Time required to produce LTL formula.

Table IV shows that LTLCodeGen consistently outperforms NL2LTL. Notably, LTLCodeGen incurs no syntactic errors, thanks to its predefined LTL operator functions. Since LLMs are better at writing code than LTL-specific syntax, this approach delivers stronger correctness guarantees. Moreover, LTLCodeGen excels in sequential and revisiting tasks, where strict ordering challenges the baseline. Expressing constraints in code appears more manageable for LLMs.

However, LTLCodeGen requires longer inference times since it generates complete executable code, including comments and explanations, while NL2LTL produces only a single-line LTL formula. Our ablation study (Section V-D) confirms that removing explanations or comments reduces generation length and accelerates inference.

D. Ablation Study

We examine three variations of our method on the same 45 real-world tasks: (1) the full version, (2) a version without task explanations, and (3) a version without line comments in the code. Each variant uses both GPT-4o and GPT-4o-mini [1] to evaluate the impact of model size.

1) No Explanation: As shown in Table IV, removing task explanations significantly reduces Tier 3 performance and noticeably harms Tier 4. While Tiers 1 and 2 remain relatively straightforward, the revisiting and sequencing in Tier 3 become error-prone without explanations, and Tier 4's contextual ambiguity exacerbates this issue. These findings demonstrate that providing explanations yields a clear net benefit across both GPT-4o and GPT-4o-mini.

2) No Comment: Eliminating line comments has minimal impact on overall performance, likely because LLMs rely more on task explanations than on code annotations. However, GPT-4o shows a slight performance drop in Tier 3, suggesting that comments can clarify complex temporal logic. In contrast, GPT-4o-mini sometimes improves when comments are removed, indicating that extraneous text may add noise for smaller models.

3) Model Size and Performance: Switching from GPT-4o to GPT-4o-mini results in a performance decline, especially for Tier 4 tasks, reinforcing that smaller models struggle more with ambiguity. Tier 3 performance also declines for all variations, reflecting the complexity of sequencing constraints. However, Tiers 1 and 2 experience only modest drops, suggesting that larger models are generally more robust across all difficulty levels.

VI. CONCLUSION

We introduced a modular, LLM-based planning framework that translates natural language instructions into syntactically correct LTL specifications, leveraging a semantic occupancy map and a motion planning algorithm to generate optimal collision-free paths satisfying the instruction. Central to this approach is LTLCodeGen, which fully exploits large language models' proficiency in code generation to reliably produce the correct LTL. Our thorough evaluation shows that with the same LLM, LTLCodeGen outperforms the

baselines in generating syntactically correct LTL formulas for varying instruction complexity. Simulation and real-world experiments suggest that LTLCodeGen plays a key role in supporting our system to robustly plan the task. These results indicate that an LLM-based framework should take what LLMs are good at into account instead of simply using prompts.

REFERENCES

[1] OpenAI, "GPT-4 Technical Report," *arXiv preprint: arXiv 2303.08774*, 2024.

[2] H. Touvron, L. Martin, *et al.*, "Llama 2: Open Foundation and Fine-Tuned Chat Models," *arXiv preprint: arXiv 2307.09288*, 2023.

[3] DeepSeek-AI, "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning," *arXiv preprint: arXiv 2501.12948*, 2025.

[4] B. Ichter, A. Brohan, *et al.*, "Do As I Can, Not As I Say: Grounding Language in Robotic Affordances," in *Conference on Robot Learning*, 2023.

[5] Z. Mandi, S. Jain, and S. Song, "RoCo: Dialectic Multi-Robot Collaboration with Large Language Models," in *IEEE International Conference on Robotics and Automation*, 2024.

[6] M. U. Nasir, S. James, and J. Togelius, "GameTraversalBenchmark: Evaluating Planning Abilities Of Large Language Models Through Traversing 2D Game Maps," *Advances in Neural Information Processing Systems*, 2024.

[7] Z. Dai, A. Asgharivaskasi, T. Duong, S. Lin, M.-E. Tzes, G. Pappas, and N. Atanasov, "Optimal Scene Graph Planning with Large Language Model Guidance," in *IEEE International Conference on Robotics and Automation*, 2024.

[8] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, "Code as Policies: Language Model Programs for Embodied Control," in *IEEE International Conference on Robotics and Automation*, 2023.

[9] X. Zhao, M. Li, C. Weber, M. B. Hafez, and S. Wermter, "Chat with the Environment: Interactive Multimodal Perception Using Large Language Models," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2023.

[10] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg, "ProgPrompt: Generating Situated Robot Task Plans using Large Language Models," in *IEEE International Conference on Robotics and Automation*, 2023.

[11] S. S. Kannan, V. L. N. Venkatesh, and B.-C. Min, "SMART-LLM: Smart Multi-Agent Robot Task Planning using Large Language Models," *arXiv preprint: arXiv 2309.10062*, 2024.

[12] Z. Ravichandran, V. Murali, M. Tzes, G. J. Pappas, and V. Kumar, "SPINE: Online Semantic Planning for Missions with Incomplete Natural Language Specifications in Unstructured Environments," *arXiv preprint: arXiv 2410.03035*, 2024.

[13] T. Silver, S. Dan, K. Srinivas, J. B. Tenenbaum, L. Kaelbling, and M. Katz, "Generalized Planning in PDDL Domains with Pretrained Large Language Models," in *AAAI/IAAI/EAAI*, 2024.

[14] S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. G. Banerjee, S. Teller, and N. Roy, "Approaching the Symbol Grounding Problem with Probabilistic Graphical Models," *AI Magazine*, 2011.

[15] H. Wang, G. Gonzalez-Pumariega, Y. Sharma, and S. Choudhury, "Demo2Code: From Summarizing Demonstrations to Synthesizing Code via Extended Chain-of-Thought," *arXiv preprint: arXiv 2305.16744*, 2023.

[16] S. Kambhampati, K. Valmeekam, L. Guan, M. Verma, K. Stechly, S. Bhambri, L. Saldyt, and A. Murthy, "LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks," *arXiv preprint: arXiv 2402.01817*, 2024.

[17] K. Valmeekam, M. Marquez, S. Sreedharan, and S. Kambhampati, "On the Planning Abilities of Large Language Models : A Critical Investigation," *arXiv preprint: arXiv 2305.15771*, 2023.

[18] W. Huang, F. Xia, *et al.*, "Inner Monologue: Embodied Reasoning through Planning with Language Models," *arXiv preprint: arXiv 2207.05608*, 2022.

[19] S. Huang, Z. Jiang, H. Dong, Y. Qiao, P. Gao, and H. Li, "Instruct2Act: Mapping Multi-modality Instructions to Robotic Actions with Large Language Model," *arXiv preprint: arXiv 2305.11176*, 2023.

[20] A. Asgharivaskasi and N. Atanasov, "Semantic Octree Mapping and Shannon Mutual Information Computation for Robot Exploration," *IEEE Transactions on Robotics*, 2023.

[21] G. Jocher, J. Qiu, and A. Chaurasia, "Ultralytics YOLO," Jan. 2023. [Online]. Available: https://github.com/ultralytics/ultralytics

[22] J. Pan, G. Chou, and D. Berenson, "Data-Efficient Learning of Natural Language to Linear Temporal Logic Translators for Robot Task Specification," in *IEEE International Conference on Robotics and Automation*, 2023.

[23] E. M. Bender and A. Koller, "Climbing towards NLU: On Meaning, Form, and Understanding in the Age of Data," in *Annual Meeting of the Association for Computational Linguistics*, 2020.

[24] R. Gubelmann, "Pragmatic Norms Are All You Need – Why The Symbol Grounding Problem Does Not Apply to LLMs," in *Conference on Empirical Methods in Natural Language Processing*, Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds. Association for Computational Linguistics, 2024.

[25] Y. Chen, J. Arkin, C. Dawson, Y. Zhang, N. Roy, and C. Fan, "AutoTAMP: Autoregressive Task and Motion Planning with LLMs as Translators and Checkers," in *IEEE International Conference on Robotics and Automation*, 2024.

[26] Y. Chen, R. Gandhi, Y. Zhang, and C. Fan, "NL2TL: Transforming Natural Languages to Temporal Logics using Large Language Models," *arXiv preprint: arXiv 2305.07766*, 2024.

[27] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray, "TuLiP: a Software Toolbox for Receding Horizon Temporal Logic Planning," in *International Conference on Hybrid Systems: Computation and Control*, 2011.

[28] C. I. Vasile and C. Belta, "Sampling-based Temporal Logic Path Planning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013.

[29] J. Fu, N. Atanasov, U. Topcu, and G. J. Pappas, "Optimal Temporal Logic Planning in Probabilistic Semantic Maps," in *IEEE International Conference on Robotics and Automation*, 2016.

[30] I. Armeni, Z.-Y. He, J. Gwak, A. R. Zamir, M. Fischer, J. Malik, and S. Savarese, "3D Scene Graph: A Structure for Unified Semantics, 3D Space, and Camera," in *IEEE International Conference on Computer Vision*, 2019.

[31] A. Rosinol, A. Gupta, M. Abate, J. Shi, and L. Carlone, "3D Dynamic Scene Graphs: Actionable Spatial Perception with Places, Objects, and Humans," in *Robotics: Science and Systems*, 2020.

[32] A. Rosinol, A. Violette, M. Abate, N. Hughes, Y. Chang, J. Shi, A. Gupta, and L. Carlone, "Kimera: From SLAM to Spatial Perception with 3D Dynamic Scene Graphs," *The International Journal of Robotics Research*, 2021.

[33] N. Hughes, Y. Chang, and L. Carlone, "Hydra: A Real-time Spatial Perception System for 3D Scene Graph Construction and Optimization," in *Robotics: Science and Systems*, 2022.

[34] S. Amiri, K. Chandan, and S. Zhang, "Reasoning with Scene Graphs for Robot Planning under Partial Observability," *IEEE Robotics and Automation Letters (RAL)*, 2022.

[35] A. Zeng, M. Attarian, *et al.*, "Socratic Models: Composing Zero-Shot Multimodal Reasoning with Language," *arXiv preprint: arXiv 2204.00598*, 2022.

[36] K. Chen, B. T. Lopez, A.-a. Agha-mohammadi, and A. Mehta, "Direct LiDAR Odometry: Fast Localization With Dense Point Clouds," *IEEE Robotics and Automation Letters*, 2022.

[37] A. Pnueli, "The Temporal Semantics of Concurrent Programs," *Theoretical Computer Science*, 1981.

[38] O. Kupferman and M. Y. Vardi, "Model Checking of Safety Properties," *Formal Methods in System Design*, 2001.

[39] A. Duret-Lutz, E. Renault, *et al.*, "From Spot 2.0 to Spot 2.10: What's New?" in *International Conference on Computer Aided Verification (CAV)*, 2022.

[40] P. Wolper, M. Y. Vardi, and A. P. Sistla, "Reasoning about infinite computation paths," in *Annual Symposium on Foundations of Computer Science*, 1983.

[41] J. Doran and D. Michie, "Experiments with the Graph Traverser Program," *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 1966.