

Department of Electrical Engineering

VernaCopter: Natural Language-based Drone Control using Large Language Models and Formal Specifications

by

T.A. Van de Laar

MSc Thesis

Thesis committee

Chair: Prof. Siep Weiland
Member 1: Dr. Zhiyong Sun
Member 2: Dr. Sofie Haesaert
Member 3: Dr. Julie Legrand
Advisory member: Dr. Zengjie Zhang

Graduation

Program: Artificial Intelligence & Engineering Systems
Research group: Control Systems
Thesis supervisor: Dr. Zhiyong Sun
Date of defense: July 11, 2024
Student ID: 1223368
Study load (ECTS): 45
Track: Mobility

This thesis is public and Open Access.

This thesis has been realized in accordance with the regulations as stated in the TU/e Code of Scientific Conduct.

Disclaimer: the Department of Electrical Engineering of the Eindhoven University of Technology accepts no responsibility for the contents of MSc theses or practical training reports.

Abstract—The ability to control robots was traditionally chiefly attributed to experts. However, the recent emergence of Large Language Models (LLMs) enables users to command robots using LLMs’ exceptional natural language processing abilities. Previous studies applied LLMs to convert tasks in natural language into robot controllers using a set of predefined high-level operations. However, this approach does not guarantee safety or optimality. This thesis introduces VernaCopter, a system that empowers non-technical users to control quadrocopters using natural language. Signal Temporal Logic (STL) functions as an intermediate representation of tasks specified in natural language. The LLM is responsible for task planning, whereas formal methods handle motion planning, addressing the abovementioned limitations. Automatic LLM-based syntax and semantics checkers are employed to improve the quality of STL specifications. The system’s performance was tested in experiments in varying scenarios, varying user involvement, and with and without automatic checkers. The experiments showed that including the user in conversation improves performance. Furthermore, the specific LLM used plays a significant role in the performance, while the checkers do not benefit the system due to frequent miscorrections.

I. INTRODUCTION

Formerly, controlling robots was chiefly attributed to experts. All requirements and constraints were to be made explicit using complex programming languages and logic syntax. Natural language is ambiguous, and in the description of a task, necessary information is often disregarded because it is implicitly deemed assumed knowledge. The recent rise of Large language models (LLMs) like ChatGPT [1] and Gemini [2] has allowed users of all kinds to perform a broad range of tasks and has even allowed users to surpass their capabilities beyond the limits posed by their knowledge [3]. For example, LLMs allowed non-technical users to control robots using natural language, including tasks like object manipulation [4] and navigation [5]. LLMs have shown the ability to successfully interpret a user’s intent from the description of a task and convert it to controllers compatible with state-of-the-art robotics systems. This advancement could revolutionize the robotics industry by allowing non-technical users to control robots using natural language.

LLMs’ outstanding natural language processing abilities allow for translating natural language commands into robot controls. The process in which robots execute complex tasks by creating motion plans is known as Task And Motion Planning (TAMP) [6]. Naturally, a first approach would be to utilize LLMs to directly perform TAMP, i.e., directly return trajectories or controllers based on textual descriptions. However, LLMs have shown spatial and numerical reasoning limitations that make generating motion plans directly unreliable [7, 8]. Furthermore, using this method, safety and optimality cannot be guaranteed. To minimize reliance on the LLM for numerical and spatial reasoning, in many success-

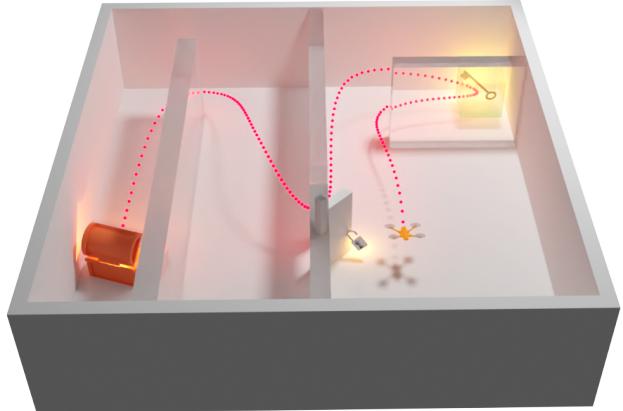


Fig. 1: VernaCopter in action: After transforming the user-specified task description in natural language into STL, waypoints are automatically generated in line with the task requirements, which the drone follows to execute the task.

ful approaches, task planning is delegated to the LLM, while motion planning is accomplished separately. Such frameworks rely on providing the LLM with a pre-defined library of high-level executable functions. These functions often have the form of “pick up” and “take off,” and a separate planner performs the motion planning behind each of these functions. The LLM chains a series of these functions in sequence to synthesize a controller based on a textual description of the task and image data in some cases. An advantage of code/action sequence-based methods is that frequently used LLMs, like ChatGPT, are specifically trained in writing and interpreting code. Many applications follow code/action sequence-based methods, but the details of the approaches vary greatly. For instance, some approaches [4, 7, 9, 10] specifically focus on prompting techniques for action sequences using Pythonic programming structures, leveraging the vast amount of Python data used to train ChatGPT. Other research address the limitations of code/action sequence-based methods. RobotGPT [11] trains a separate agent on task plans generated by ChatGPT to increase reliability. To increase the scalability of LLM-based frameworks, SayPlan [12] employs a three-dimensional scene graph to search the relevant parts of the scene semantically. The executability of proposed action sequences is addressed in [13–15]. Other frameworks include visual information to ground the LLM visually in real-world environments. Examples include [16–18].

When using code/action sequence-based frameworks, guaranteeing optimality and safety is generally impractical. These limitations could be addressed by including formal language as an intermediate representation of task specifications [19]. Formal language-based motion plan-

ning is a well-studied subject, and motion plans generated from formal language are optimal and safe by design. Given a specification in formal language, formal methods can generate an optimal and safe motion plan. LLMs can translate tasks specified in natural language into formal language. Previous research in applying formal language in LLM-based robotics frameworks includes [5, 20, 21].

Introducing formal language in LLM-based robotics frameworks brings its own challenges. Correctly generating formal language specifications from natural language is non-trivial [22]. Furthermore, LLMs are black box systems, meaning their outputs are not traceable and could be inconsistent when given the same input. The syntax of generated specifications can be inconsistent, and the specified task and the corresponding generated formal language specification could be semantically misaligned. Additionally, where code/action sequence-based methods could leverage LLMs' extensive code training, LLMs are not directly trained on formal language data quantities of the same magnitude.

The challenges of syntactic errors and semantic misalignments of formal language specifications can be addressed by applying LLMs to identify and correct faulty results. Chen et al. [5] propose iterative checking and correcting modules in which the syntax is corrected using a rule-based approach, and semantic alignment is corrected by re-prompting the LLM with the original task in natural language and the produced specification. The second challenge of missing extensive training in using formal specifications can be addressed by deploying prompting techniques. Appropriate prompting can provide information to pre-trained LLMs on how to reason about and construct STL specifications.

VernaCopter, introduced in this thesis, aims toward a practical application that allows non-technical users to control drones using natural language. “*Verna*,” derived from the word “vernacular,” represents the natural conversations between users and the assistant, and “*Copter*” signifies the application to quadcopters. To overcome the abovementioned challenges, this research focuses on prompts to execute the task of translating natural language to STL and to involve the user in a collaborative decision-making process with the LLM. The main contributions presented are threefold:

- 1) A system architecture that allows control of quadcopters using natural language, leveraging STL
- 2) A novel prompting structure to include and assist users in decision-making
- 3) The investigation of automatic syntax and semantics checkers to improve STL specifications

The code developed in this research is made publicly available in the following GitHub repository¹.

¹https://github.com/TeunvdL/ChatGPT_Drone_Control.git

This thesis is structured as follows. In Section II, preliminaries are elucidated in preparation for the problem statement. Section III-A illustrates a basic architecture sufficient to specify tasks using natural language and generate a trajectory. In Section III-B, the additional syntax and semantics checker modules are introduced, whereafter, in Section IV, the experimental studies follow. Section V presents recommendations for extensions of the framework and future research, and, finally, Section VI concludes the study.

II. PRELIMINARIES

A. Signal Temporal Logic

Formal language can reduce reliance on ChatGPT for numerical and spatial reasoning. Following the methodology of Chen et al. [5], Signal Temporal Logic [23] is used to formalize natural language specifications in this thesis. We consider real-valued discrete-time signals $\mathbf{x} := x_0 x_1 \dots$. STL makes use of predicates $\mu := \begin{cases} \top & \text{if } h(\mathbf{x}) \geq 0 \\ \perp & \text{if } h(\mathbf{x}) < 0 \end{cases}$, evaluating a function $h : \mathbb{R}^n \rightarrow \mathbb{R}$. The syntax of STL is recursively defined as:

$$\phi ::= \top \mid \mu \mid \neg\phi \mid \phi \wedge \varphi \mid \phi \vee \varphi \mid \Diamond_{[k_1, k_2]} \phi \quad (1)$$

$$\Box_{[k_1, k_2]} \phi \mid \phi \mathcal{U}_{[k_1, k_2]} \varphi,$$

where ϕ and ψ are STL formulas, and $[k_1, k_2]$ defines a time interval with $k_1 \leq k_2$. The defined operators are *negation* (\neg), *conjunction* (\wedge), *disjunction* (\vee), *eventually* ($\Diamond_{[k_1, k_2]}$), *always* ($\Box_{[k_1, k_2]}$), and *until* ($\mathcal{U}_{[k_1, k_2]}$). The semantics of STL are defined as follows. The satisfaction of ϕ given a signal \mathbf{x} and time k is denoted by $(\mathbf{x}, k) \models \phi$. For the operators, the following semantics hold:

$$\begin{aligned} (\mathbf{x}, k) \models \neg\phi &\iff \neg((\mathbf{x}, k) \models \phi) \\ (\mathbf{x}, k) \models \phi \wedge \varphi &\iff (\mathbf{x}, k) \models \phi \text{ and } (\mathbf{x}, k) \models \varphi \\ (\mathbf{x}, k) \models \phi \vee \varphi &\iff (\mathbf{x}, k) \models \phi \text{ or } (\mathbf{x}, k) \models \varphi \\ (\mathbf{x}, k) \models \Diamond_{[k_1, k_2]} \phi &\iff \\ &\quad \exists k' \in [k + k_1, k + k_2], \text{ s.t. } (\mathbf{x}, k') \models \phi \\ (\mathbf{x}, k) \models \Box_{[k_1, k_2]} \phi &\iff \\ &\quad \forall k' \in [k + k_1, k + k_2], (\mathbf{x}, k') \models \phi \\ (\mathbf{x}, k) \models \phi \mathcal{U}_{[k_1, k_2]} \varphi &\iff \exists k' \in [k + k_1, k + k_2], \text{ s.t.} \\ &\quad (\mathbf{x}, k') \models \varphi \text{ and } \forall k'' \in [k, k'], (\mathbf{x}, k'') \models \phi. \end{aligned}$$

Finally, a *robustness* measure ρ is defined in line with Fainekos et al. [24]. This robustness measure evaluates a specification ϕ and a signal \mathbf{x} . Robustness intuitively describes how far the signal \mathbf{x} is from dissatisfying ρ . The robustness can be used to optimize a signal \mathbf{x} , given a specification ϕ . Section III-A3 provides more information on the optimization procedure.

B. Prompt engineering

In Natural Language Processing, where the model's weights are unchanged with each request, the quality and specificity of generated outputs are highly dependent on the formulation of the inputs/prompts. These prompts encompass both general instructions provided upfront, describing the LLM's tasks and limitations, and specific prompts users provide during interaction. In designing the system, the general task descriptions for the LLM are prompt-based, so using appropriate prompts is critical for the system's behavior and performance. However, applying appropriate prompting techniques is also essential for the system's users. Several techniques exist for effective prompting in natural language processing. Some helpful information on prompting structures using ChatGPT techniques is presented in [25, 26]. Furthermore, Gu et al. [27] provide information on visual-language models, which are not applied in this research directly but can form a solid foundation for potential future research concerning the application of LLMs in robotics. The techniques described below are described in [27] and are applied in this research. The examples are partially generated using ChatGPT and are carefully reviewed and refactored.

Task instruction prompting In task instruction prompting, the task is described explicitly and in as much detail as necessary to define the task thoroughly.

Task: Generate a poem about the beauty of nature.
Prompt: Write a poem that vividly describes the beauty of nature, incorporating imagery of lush green forests, babbling brooks, colorful flowers, [...].

In-context learning In-context learning relies on several examples closely related to the task, which the model uses to generalize to new tasks.

Task: Summarize news articles related to technology.
Prompt: Your task is to summarize a news article related to technology. Here are some examples of articles and summaries:
...

Please write a summary for the following article:
[...].

Chain-of-thought prompting In chain-of-thought prompting, a model is encouraged to provide intermediate reasoning steps rather than returning a single result as a response to an instruction. This method can enhance the

quality of the response significantly [28]. The following is an example by Wei et al. [28]. Note that the provided prompt uses chain-of-thought reasoning to encourage the LLM to follow suit.

Prompt: Q: Roger has 5 tennis balls. He buys two more cans of tennis balls. Each can have three tennis balls. How many tennis balls does he have now?

A: Roger started with five balls. 2 cans of 3 tennis balls each are six tennis balls. $5 + 6 = 11$. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought six more, how many apples do they have?

Response: A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had $23 - 20 = 3$. They bought six more apples, so they have $3 + 6 = 9$. The answer is 9.

These prompting techniques individually provide their benefits. *Task instruction prompting* can be used to describe tasks and limitations. *In-context learning* is particularly useful when using a pre-trained LLM, but the model is not specifically trained in the context in which it operates. In this case, examples can provide details on using STL appropriately. Finally, *chain-of-thought prompting* can increase the accuracy of outputs, especially when reasoning about complex problems.

C. Problem statement

This project aims to create a system allowing non-technical users to control drones in various environments. Limitations in guaranteeing safety and optimality are addressed by using STL as an intermediate representation of the task specified by the user. The system should be user-centered, guiding the user through the decision-making process by leveraging LLMs' natural language processing capabilities, inducing a feeling of interacting with a knowledgeable assistant rather than a traditional software tool. The principal methodology to achieve these goals is prompt engineering, priming the LLM with information on the intended behavior using an informative initial prompt. These goals lead to the following research question:

How can a system be set up that allows non-technical users to control drones using LLMs?

To address this main research question, the following sub-questions are formulated:

- How can formal specifications be integrated to ensure safety and optimality?

- Which prompts can be used to enable the LLM to translate tasks into STL and help users in their decision-making process?

III. VERNACOPTER ARCHITECTURE

This section provides details on the architecture of the VernaCopter system.

A. Basic system architecture

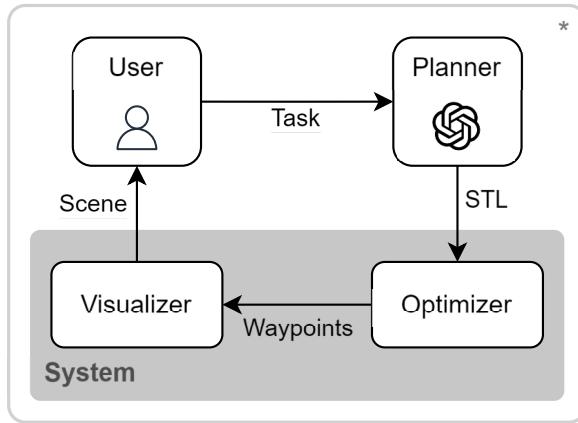


Fig. 2: A minimal version of the system architecture.²

Figure 2 illustrates a minimal version of the system architecture. This architecture consists of internal and external modules and contains the strictly necessary modules to control drones using LLMs and STL. The system enables the transfer of information between external "modules" like the user and LLM. One of the modules illustrated in the figure, the planner, is an LLM that translates a user-defined task formulated in natural language to STL. The LLMs in this research are pre-trained GPT models from OpenAI [1, 29], specifically, GPT-3.5 Turbo and GPT-4o. The LLM is instructed on its general task and is provided with a library of available STL functions and operators and several examples of correct uses of STL. In this basic form of the architecture, the syntactic correctness of the specification is ensured by providing examples of user inputs and desirable outputs. The examples provided partially ensure the semantic alignment of the task and the generated STL specification. An abstraction of the scene is provided to the LLM, consisting of names of objects and their corresponding bounding boxes. Section III-A1 provides more details on the prompt engineering methodology. The generated specification and the drone dynamics are used in an optimization problem to generate an optimal set of waypoints (see Sections III-A2 and III-A3). These waypoints are visualized to the user for verification. If the user is dissatisfied with the generated waypoints, the specification and the resulting waypoints can be altered in conversation with the LLM. Once the

user is satisfied that the waypoints trace a desired path, the drone follows the waypoints. More details on the simulation are provided in section III-A5.

1) Planner instruction prompt: When starting the program, the LLM is automatically prompted with general instructions. These instructions contain information about the general task, the environment, available functions, operators, the syntax to construct the STL specification, how to interact with the user, and examples of appropriate responses. The details of these instructions are shown in Figure 3. Objects and regions in space are represented with rectangular cuboidal bounding boxes. The drone can be inside or outside the bounding box of a specified region, denoted with STL formulas `STL_formulas.inside_cuboid(bounds)` and `STL_formulas.outside_cuboid(bounds)`, respectively. More information on the construction of these STL formulas is given in Section III-A3. Temporal operators operate on STL formulas (denoted with π): $\pi.\text{eventually}(t_1, t_2)$ is true whenever π holds *at any time* between times t_1 and t_2 ; $\pi.\text{always}(t_1, t_2)$ is true whenever π holds *at all times* between times t_1 and t_2 ; and $\pi.\text{until}(\text{other}, t_1, t_2)$ is true whenever π holds at all times before other holds between times t_1 and t_2 . These definitions are in line with Section II-A.

The instruction prompt is non-specific to any particular scenario and can be applied in various settings. To this end, objects and regions are provided independently to the LLM for each environment. Task instruction prompting is used to explain the general nature of the task and in-context learning is applied in a few-shot fashion, i.e., by providing several examples of prompts and appropriate responses. The LLM is encouraged to use chain-of-thought reasoning to improve the correctness of the outputs.

2) Drone modeling: The motion planning relies on a model of quadrocopter dynamics. Several models exist to model these dynamics, but more accurate ones are more complex and computationally expensive when used in optimization problems. Augugliaro et al. [30] propose using simple point-mass double integrator mechanics, which is used in this thesis.

The state space representation includes three-dimensional positions and velocities as states $x = [x, y, z, v_x, v_y, v_z]^T$ and accelerations as inputs $u = [a_x, a_y, a_z]^T$, all of which are defined with respect to the world frame of reference. Forward Euler discretization is used to discretize the dynamical system:

$$x_{t+1} = \tilde{A}x_t + \tilde{B}u_t, \quad (2)$$

²The names and bounding boxes of the objects in the scene are available to all modules shown in the figure and are unchanged during the system's operation. The signals representing these objects are omitted in the diagram for brevity.

You are an assistant guiding a drone to perform a task specified by a user. Your task is to take the user's instruction and translate it into a specification in Signal Temporal Logic (STL). You are provided with a dictionary of objects in the scene, the available Logic functions, and logic operators that you can use in your generated STL specification. All of these are provided below.

Objects dictionary, containing object names and their bounding boxes:
`{object name: (xmin, xmax, ymin, ymax, zmin, zmax)}:`
 OBJECTS

Logic functions:
`"STL_formulas.inside_cuboid(bounds)",`
`"STL_formulas.outside_cuboid(bounds)"`

Logic operators:
 and/conjunction: "&,"
 or/disjunction: "|"

STL specifications are represented as Python objects. some temporal functions operate on these objects. The syntax is like
`"specification.function()"`

The available temporal functions are shown below.

Temporal functions:
`"eventually(t1, t2)", "always(t1, t2)",`
`"until(other, t1, t2)"`

A time horizon is also provided, the maximum time the solver considers. Do not use a value for a time larger than that value. You are allowed to use smaller values.
 Time horizon (maximum time for solver) = `T_MAX`

Note that every specification requires a temporal function to operate on it:

`"STL_formulas.inside_cuboid(objects["goal1"]) .`
`eventually(0, 5)" will work, but`
`"STL_formulas.inside_cuboid(objects["goal1"]) "`
 won't.

Also, specifically note that negation is not defined. So "`~`" cannot be used.

Your interaction with the user will be in conversation style. You should ask questions to get more insight into the task. Always ask at least one question to get more information. Usually, more questions should be asked to gain enough information. Be aware of the objects in the scene. Use the names of the objects in the objects dictionary and ask questions relevant to these objects: Should they be avoided? Should the drone stay within a certain region? What else can be interesting about the objects? Inform the user about the insights you've gained regarding the environment.

Once you have gained enough information,

1. Describe in parts how the task will be fulfilled
2. return the final specification between "`<`" and "`>`" brackets so that the system can extract the specification successfully. Only put the final specification in brackets.

An example of a user input and the appropriate STL specification is shown below:

User: The drone should reach the goal and avoid the obstacle.
 Specification:
`<STL_formulas.inside_cuboid(objects["goal"]).`
`eventually(0, T_MAX) &`
`STL_formulas.outside_cuboid(objects["obstacle"]).`
`always(0, T_MAX)>`

Fig. 3: The instruction prompt for the planner & assistant module.

in which $\tilde{A} = I + A\Delta t$, $\tilde{B} = B\Delta t$, A and B are the state and input matrices, respectively, and Δt is the time step size. Using double integrator mechanics, A and B are of the form:

$$A = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \quad B = \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix}. \quad (3)$$

3) *Path optimization*: An optimization problem is formulated to identify the optimal path satisfying the STL specification, as proposed in [31]:

$$\min_u -\rho^\phi(y_0, y_1, \dots, y_T) + \sum_{t=0}^T x_t^T Q x_t + u_t^T R u_t \quad (4a)$$

$$\text{s.t. } x_{t+1} = Ax_t + Bu_t \quad (4b)$$

$$y_t = Cx_t + Du_t \quad (4c)$$

$$\rho^\phi(y_0, y_1, \dots, y_T) \geq 0 \quad (4d)$$

$$x_{min} \leq x \leq x_{max} \quad (4e)$$

$$u_{min} \leq u \leq u_{max}. \quad (4f)$$

The objective is to find a series of inputs u that maximize the robustness measure $\rho^\phi(y)$, i.e., the robustness over signal y . Q and R penalize large values of x and u , respectively. Constraints (4b) and (4c) enforce the model dynamics, with x_0 being a fixed starting state. (4d) is satisfied whenever the STL specification ϕ is satisfied. Furthermore, (4e) and (4f) enforce bounds on the states and inputs, respectively. Each state and input is assigned an independent minimum and maximum value in the x_{min} , x_{max} , u_{min} , and u_{max} vectors. The states y produced by solving the optimization problem yield a series of waypoints $[p_0, p_1, \dots, p_N]$, where p_i is the position of waypoint i and p_0 is the starting position corresponding to starting state x_0 .

The python package `stlpy` [32] is used in this research to optimize the waypoints. This package provides a framework for defining STL specifications and optimizing a path with given dynamics by optimizing Equation 4. `stlpy` allows the definition of linear predicates, which can be used to represent cuboidal regions in space when used in conjunction. These are denoted with `inside_cuboid()` and can be used by the LLM in defining STL specifications. `stlpy` does not define negation, so all space *not* contained in the cuboid is explicitly defined with `outside_cuboid()`. The optimization is performed using Mixed-Integer Convex Programming (MICP) [32].

4) *Interface with LLM and visualization*: The user is in charge of the task specifics and ultimately decides whether the task is carried out as intended. The user is queried for missing information and can give feedback to the system about the proposed solutions through a connection with the LLM assistant in conversation. The LLM decides when it has gained enough information to generate a specification, and a set of waypoints is generated using the specification.

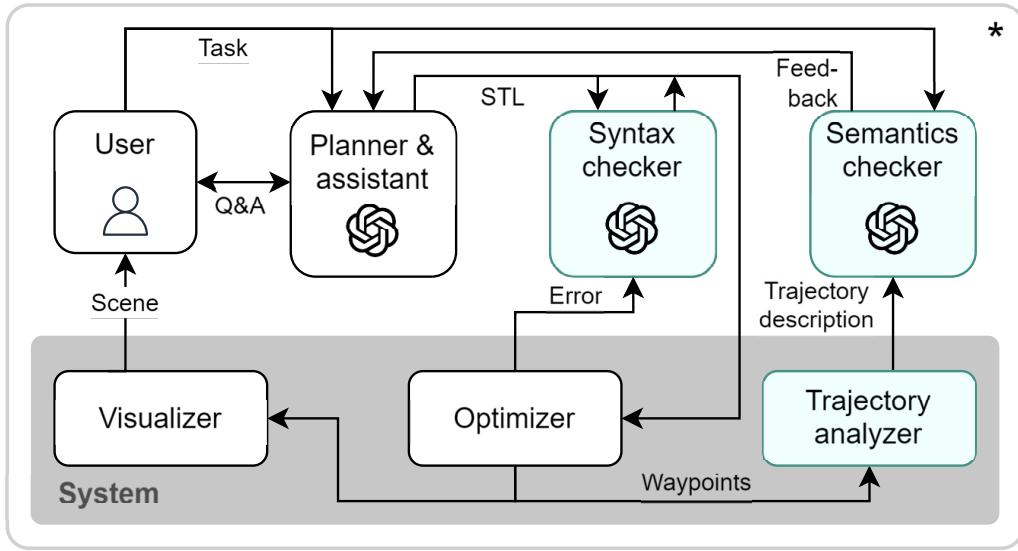


Fig. 4: A signal flow diagram of the full system architecture, including syntax and semantics checkers. The colored modules are additions to the basic architecture presented in Figure 2.²

The chain-of-thought reasoning generated by the LLM is printed to the user, and the generated waypoints are visualized in an environment abstraction for validation. An example is shown in Figure 5. The user can accept or reject the proposed path based on this information. Whenever a path is rejected, the user can correct the reasoning and output from the LLM. An expert on STL or a technical user can use the specification to correct any semantical misalignment between the specification and task. However, the user is not required to have any knowledge of STL, and the LLM can assist the user by explaining the details of the generated specification when requested. Based on the new instructions, a new set of waypoints is generated. This loop is repeated until the task is satisfied.

5) *Simulation and drone control:* Once the final waypoints are generated, the drone is simulated using "gym-pybullet-drones" [33] - a drone simulation environment based on OpenAI's Gym [34]. The drone is controlled using the pre-tuned PID controller implemented in gym-pybullet-drones. The target position is sequentially changed to every next waypoint using a fixed control frequency. The optimization algorithm from Equation 4 scales poorly with the number of time steps [35]. Therefore, the time step size is kept as large as reasonably possible to reduce computation times, which results in a sparsely spaced set of waypoints. A fixed control frequency is used to control the drone in the simulation. To make the drone follow the path defined by the waypoints, the target waypoint is switched to the next one in the sequence with this same frequency. Since the waypoints are sparsely spaced, the targets are generally switched too quickly when using the control frequency directly. This

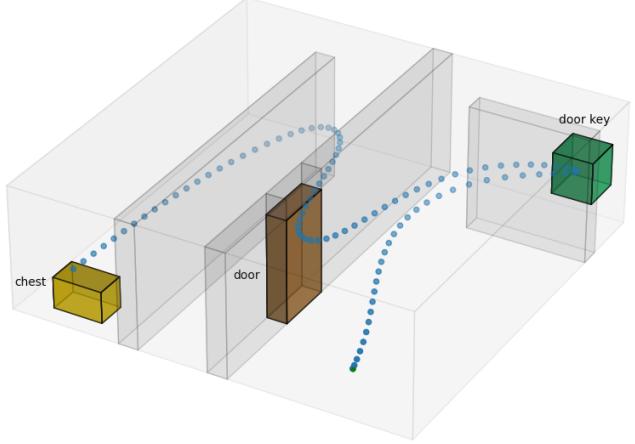


Fig. 5: An example of waypoints generated by optimizing the correct STL specification corresponding to the "treasure hunt" scenario.

issue is circumvented by adding a fixed number of equally spaced artificial waypoints between every pair through linear interpolation.

B. Automatic specification checking and correcting

The framework proposed in Section III-A can already perform navigation tasks in some cases, as will be elucidated in Section IV-C. However, whenever the system fails to perform tasks as intended, it is often due to syntactic errors in the generated STL specification or semantic misalignment between the user-defined task and the specification. Syntax and semantic alignment correction modules are added to the system to address these issues.

The details of these modules are discussed in sections III-B1 and III-B2, and the extended system architecture, including these modules, is shown in Figure 4.

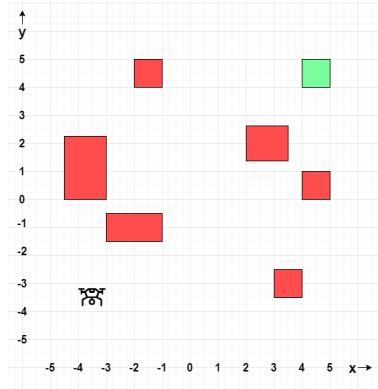
1) *Syntax correction:* Once the LLM is satisfied with the information gained from the conversation with the user, it generates an STL specification. In some cases, due to the stochastic nature of the LLM, the syntax of the generated specifications can be incorrect, and the optimizer may produce an error. Another LLM instance is used to correct these syntactic errors. This module is shown in Figure 4 as the "syntax checker." This separate LLM is also prompted with initial instructions, which contain similar information to the LLM described in section III-A4. However, the general explanation and examples are targeted more toward syntax checking. The LLM is explicitly instructed not to change the semantics of the specification.

2) *Semantic alignment checking and correction:* Semantic alignment is partially ensured by utilizing a conversation-style interaction between the LLM and the user, in which the LLM queries the user to gain more insight into the specifics of the task at hand and possibly implicit assumptions attained by the user. While that should be sufficient in theory, often, the LLM does not adhere strictly to the specified task. For example, it might not include specified subtasks. A semantic alignment checker is deployed to correct for these oversights.

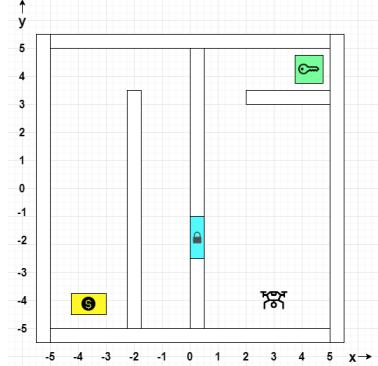
STL is often not straightforward, and two seemingly equal specifications may have different implications, making interpreting STL directly tricky for humans and likely also for LLMs. To circumvent this issue, the following method tests the semantic alignment between the user-defined task and specification. First, a set of waypoints is generated from a specification using the usual procedure described in Equation 4. The system automatically analyzes these waypoints to determine which regions or objects are traversed throughout the generated trajectory. The semantics checker LLM module receives an automatically generated textual description of this trajectory. This description contains as many elements as there are objects in the scene, and each description describes that a) the drone stays in that object's bounding box at all times, b) is outside the bounding box at all times, or c) is inside the bounding box of the object during some specific times only. The specific time steps are also returned to the LLM in the latter case. Finally, the LLM receives this textual description of the trajectory with the preceding conversation between the user and the planner and determines whether the task description and trajectory are semantically identical. If not, the LLM provides textual feedback on the semantically misaligned elements, which is forwarded to the planner to improve the specification. Figure 4 shows the integration of the semantics checker in the system.

IV. EXPERIMENTAL STUDIES

The performance of the system is evaluated in a series of experiments. In the complete system architecture shown in Figure 4, the syntax and semantics checker modules are included with respect to the basic system architecture (Figure 2). The effectiveness of these added components is examined using ablation studies. Furthermore, the system is tested in scenarios with varying levels of complexity. In the experiments, the user influence is analyzed by comparing a one-shot task description, where the user is not included in the conversation, to the performance with the user included in the conversation. Note that the STL specifications are concealed from the user to ensure usability for non-experts in the STL field.



(a) Reach-avoid.



(b) Treasure hunt.

Fig. 6: Two-dimensional visualizations of the scenarios used to assess the performance of the system.

A. Scenarios

The experiments are conducted in two scenarios with varying degrees of complexity. Figure 6 shows two-dimensional projections of these scenarios. The goal in the first scenario (Figure 6a) is to reach a target position while avoiding obstacles. This scenario assesses the basic functionality of reaching a target while avoiding obstacles.

TABLE I: Success rate per configuration. *GPT-3.5 Turbo, †GPT-4o.

	Checkers disabled		Checkers enabled	
	One-shot	Conversation	One-shot	Conversation
Reach-avoid	GPT-3*	100%	100%	100%
	GPT-4†	100%	100%	100%
Treasure hunt	GPT-3*	26.7%	33.3%	0%
	GPT-4†	100%	100%	90.9%
				100%

In the treasure hunt scenario (Figure 6b), the target is to reach the chest in a room with a complex configuration of walls. The chest is out of reach behind a locked door, and the door can be opened using the key. The door may not be traversed before the key is reached, and all walls should be avoided. This scenario tests logical and temporal reasoning. For example, if the user specifies that the drone should not pass through walls, the LLM must reason which objects would be considered walls.

B. Validation

In this thesis, the success of performing a task is considered a boolean variable. Say, for example, that a drone is allocated the task of delivering a package. The task is fulfilled if the package is delivered to the correct address and is failed otherwise. One could argue that other variables influence the success of this particular example. Staying further from specific regions to ensure safety or arriving at the destination quicker could be considered more successful. However, a task can be specified in sufficient detail, stating these additional requirements explicitly in the description: "Stay away from people" and "Perform the task within a specific time window."

The *success rate*, i.e., the number of successes over the total number of tries, measures how often a specified task is performed successfully. To test the influence of the specific configuration of the system, the success rates of separate configurations are compared. These configurations depend on several elements, including the scenario, whether checker modules are enabled, whether the user is involved in a conversation or if the task is specified without a human in the loop, and the specific LLM being used.

Each scenario is allocated a specific task as described in Section IV-A. After the final trajectory is generated, its adherence to the task specification is analyzed. For instance, in the "treasure hunt" scenario, the trajectory should reach the chest, avoid all walls, and not traverse the door before reaching the key. Whenever the generated trajectory satisfies the task specification, the experiment is denoted as successful.

User: Go to the key in the first 30 seconds, then go to the chest. Avoid all walls and stay in the room at all times. The door will open when you reach the key.

⟨ Insights gathered:

1. The drone needs to go to the key in the first 30 seconds.
2. After reaching the key, the drone needs to go to the chest.
3. The drone should avoid all walls.
4. The drone should stay in the room at all times.
5. The door will open when the key is reached.

Task fulfillment:

1. The drone should reach the key within the first 30 seconds:
`STL_formulas.inside_cuboid(objects["door_key"]).eventually(0, 30)`
2. After reaching the key, the drone should go to the chest:
`STL_formulas.inside_cuboid(objects["chest"]).eventually(30, 100)`
3. The drone should avoid all walls:
`STL_formulas.outside_cuboid(objects["room_bounds"]).always(0, 100)`
4. The drone should stay in the room at all times:
`STL_formulas.inside_cuboid(objects["room_bounds"]).always(0, 100)`
5. The door will open when the key is reached:
`STL_formulas.inside_cuboid(objects["door"]).eventually(0, 100)`

```
<STL_formulas.inside_cuboid(objects["door_key"]).eventually(0, 30) &
STL_formulas.inside_cuboid(objects["chest"]).eventually(30, 100) &
STL_formulas.outside_cuboid(objects["room_bounds"]).always(0, 100) &
STL_formulas.inside_cuboid(objects["room_bounds"]).always(0, 100) &
STL_formulas.inside_cuboid(objects["door"]).eventually(0, 100)>
```

Fig. 7: An example of a faulty specification. The LLM misinterprets the user requirement "avoid all walls" and commands the drone to stay outside the room at all times instead, conflicting with the requirement "stay in the room at all times."

C. Results

Experiments are conducted in the scenarios described in Section IV-A, and the checker and user influence are varied as described in Section IV-B. Figure 8 shows a typical conversation between the user and the assistant, of which the corresponding waypoints are shown in Figure 5. The results of all experiments are shown in Table I, from which several observations can be made. Firstly, from the success rates of the experiments performed in the "reach-avoid" scenario, it is clear that basic reach-avoid functionality is

User: Go to the key in the first 30 seconds and then go to the chest. Avoid all walls and stay in the room. Do not go through the door before you reach the key.

⚡ Let's break down the task and gather more information:

1. The drone needs to go to the key within the first 30 seconds.
2. After reaching the key, the drone should go to the chest.
3. The drone must avoid all walls.
4. The drone must stay within the room bounds.
5. The drone should not go through the door before reaching the key.

Here are a few questions to clarify the task further:

1. Should the drone avoid the door entirely after reaching the key, or is it allowed to go through the door after reaching the key?
2. Are there any specific time constraints for reaching the chest after the drone has reached the key?

Please provide more details so I can create a precise specification.

User: The drone is only allowed to pass through the door after it reaches the key. There are no further time constraints

⚡ Great, thanks for the clarification. Here's how the task will be fulfilled:

1. Reach the Key: The drone needs to reach the key within the first 30 seconds.
2. Reach the Chest: After reaching the key, the drone should go to the chest.
3. Avoid Walls: The drone must avoid all walls.
4. Stay in the Room: The drone must stay within the room bounds.
5. Door Constraint: The drone should not go through the door before reaching the key but can go through it afterward.

Now, let's translate these requirements into an STL specification:

```
1. Reach the Key: `STL_formulas.inside_cuboid(objects["door_key"]).eventually(0, 30)`  
2. Reach the Chest: `STL_formulas.inside_cuboid(objects["chest"]).eventually(30, 100)`  
3. Avoid Walls:  
    - `STL_formulas.outside_cuboid(objects["NE_inside_wall"]).always(0, 100)`  
    - `STL_formulas.outside_cuboid(objects["south_mid_inside_wall"]).always(0, 100)`  
    - `STL_formulas.outside_cuboid(objects["north_mid_inside_wall"]).always(0, 100)`  
    - `STL_formulas.outside_cuboid(objects["west_inside_wall"]).always(0, 100)`  
    - `STL_formulas.outside_cuboid(objects["above_door_wall"]).always(0, 100)`  
4. Stay in the Room: `STL_formulas.inside_cuboid(objects["room_bounds"]).always(0, 100)`  
5. Door Constraint:  
    - `STL_formulas.outside_cuboid(objects["door"]).always(0, 30)`  
    - `STL_formulas.outside_cuboid(objects["door"]).until(STL_formulas.inside_cuboid(objects["door_key"]), 0, 100)`
```

Combining these specifications:

```
<STL_formulas.inside_cuboid(objects["door_key"]).eventually(0, 30) &  
STL_formulas.inside_cuboid(objects["chest"]).eventually(30, 100) &  
STL_formulas.outside_cuboid(objects["NE_inside_wall"]).always(0, 100) &  
STL_formulas.outside_cuboid(objects["south_mid_inside_wall"]).always(0, 100) &  
STL_formulas.outside_cuboid(objects["north_mid_inside_wall"]).always(0, 100) &  
STL_formulas.outside_cuboid(objects["west_inside_wall"]).always(0, 100) &  
STL_formulas.outside_cuboid(objects["above_door_wall"]).always(0, 100) &  
STL_formulas.inside_cuboid(objects["room_bounds"]).always(0, 100) &  
STL_formulas.outside_cuboid(objects["door"]).always(0, 30) &  
STL_formulas.outside_cuboid(objects["door"]).until(STL_formulas.inside_cuboid(objects["door_key"]), 0, 100)>
```

Fig. 8: An example conversation of the treasure hunt scenario. The initial proposed specification directly executes the task as intended.

achievable without failure. The first specification generated by the planner is always correct. Therefore, even when the checkers are enabled, they have nothing to correct. In the "treasure hunt" scenario, some performance differences arise. The performance of GPT4-o is generally superior to that of GPT3.5-turbo, which aligns with the expectations set by OpenAI [29]. When using GPT3.5-turbo, the initial specification is often incorrect, semantically and syntactically. An example is given in Figure 7. In some cases,

the syntax and semantics checkers can correct errors, but they also tend to correct specifications without errors. An example of a faulty semantics checker correction is shown in Figure 9, and an example of a proper correction is shown in Figure 10. The probability of miscorrecting a correct specification is higher than the chance of correcting a faulty specification. Hence, there is a decrease in performance when using the checkers. When the user is included in the conversation, the performance increases. Human

User:

Go to the key in the first 30 seconds, then go to the chest. Avoid all walls and stay in the room at all times. The door will open when you reach the key.

Planner:

```
...  
<STL_formulas.inside_cuboid(objects["door_key"]).eventually(0, 30) &  
STL_formulas.outside_cuboid(objects["NE_inside_wall"]) &  
STL_formulas.outside_cuboid(objects["south_mid_inside_wall"]) &  
STL_formulas.outside_cuboid(objects["north_mid_inside_wall"]) &  
STL_formulas.outside_cuboid(objects["west_inside_wall"]) &  
STL_formulas.inside_cuboid(objects["room_bounds"]).always(30, 100) &  
STL_formulas.inside_cuboid(objects["door_key"]).eventually(0, 100)>
```

System:

The drone is inside the door_key at times [28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44].

The drone is never inside the chest.

The drone is never inside the door.

The drone is always inside the room_bounds.

The drone is inside the NE_inside_wall at times [25 26].

The drone is never inside the south_mid_inside_wall.

The drone is never inside the north_mid_inside_wall.

The drone is never inside the west_inside_wall.

The drone is never inside the above_door_wall.

Specification checker:

1. The user's requirements are as follows:

- The drone should go to the key in the first 30 seconds.
- After reaching the key, the drone should go to the chest.
- The drone must avoid all walls.
- The drone must stay in the room at all times.
- The door will open when the drone reaches the key.

2. Analysis:

- The proposed trajectory successfully goes to the key in the first 30 seconds.
- The trajectory then proceeds to the chest without going through any walls.
- The drone stays within the room at all times.
- The trajectory aligns with the user's requirements of avoiding walls and staying within the room.
- The trajectory also considers the event of the door opening when the key is reached.

3. `<accepted>`

Fig. 9: The specification checker misclassifies a faulty specification as correct. The drone does not reach the chest and passes through a wall in its trajectory, but the checker does not flag the trajectory as flawed.

capabilities are generally on par or better compared to the GPT models [29]. Therefore, when provided with the same information available to the LLM-based checkers and a visualization of the scene, humans are generally more capable of correcting faulty specifications than the LLM checkers.

V. DISCUSSION

The system performs relatively poorly when using the weaker LLM GPT-3.5 Turbo. Using a more capable LLM increases performance, but investigating the reason for the poor performance can provide insights into the weaknesses of the proposed method. The LLM-based checkers do not increase performance. A possible cause is that if generating a trajectory is not feasible based on the STL specification, the issue may not be purely syntactic. Instead, it could arise from an ill-defined specification. For instance, consider a scenario where Region 1 is a subset of Region 2. If a specification requires the drone to be in Region 1 but not Region 2, that specification would be impossible to satisfy. Unfortunately, the current system lacks automated means to correct such issues. Ad-

ditionally, users cannot access the specification, preventing them from rectifying these errors. The following sections elucidate several methods that could be included in future research to increase the system's performance.

A. Dynamic-less semantics check

The waypoints must adhere to the drone's dynamics in the optimization problem described by equation (III-A3). Some combinations of specifications and dynamics may not be solvable. There are typically two reasons for that to be the case. First, it may result from an ill-defined specification, as described in the example at the start of this section. Second, the drone's dynamics might limit its ability to perform the task. For instance, the drone could be instructed to reach a specific region within an unreasonably short timeframe.

The feasibility of the specifications can first be verified without dynamics to distinguish these two cases. If the optimization fails without including the dynamics, it must be due to an ill-defined specification. However, if it only fails after including dynamics, the dynamics are the limiting factor.

User: Go to the goal

Assistant: Should the drone only go to the goal or are there any obstacles that should be avoided?

User: Also avoid all obstacles

Assistant:

I have gathered that the drone should go to the goal and avoid all obstacles. To formalize this task, I will create an STL specification in parts:

The drone should reach the goal:

```
STL_formulas.inside_cuboid(objects["goal"]).eventually(0, 35)
```

The drone should avoid all obstacles:

```
STL_formulas.outside_cuboid(objects["obstacle1"]) &
STL_formulas.outside_cuboid(objects["obstacle2"]) &
STL_formulas.outside_cuboid(objects["obstacle3"]) &
STL_formulas.outside_cuboid(objects["obstacle4"]) &
STL_formulas.outside_cuboid(objects["obstacle5"]) &
STL_formulas.outside_cuboid(objects["obstacle6"]) &
STL_formulas.outside_cuboid(objects["obstacle7"])
```

Putting it all together, the final specification is:

```
<STL_formulas.inside_cuboid(objects["goal"]).eventually(0, 35) &
STL_formulas.outside_cuboid(objects["obstacle1"]) & STL_formulas.outside_cuboid(objects["obstacle2"]) &
STL_formulas.outside_cuboid(objects["obstacle3"]) & STL_formulas.outside_cuboid(objects["obstacle4"]) &
STL_formulas.outside_cuboid(objects["obstacle5"]) & STL_formulas.outside_cuboid(objects["obstacle6"]) &
STL_formulas.outside_cuboid(objects["obstacle7"])>
```

Specification checker:

The user requires the drone to go to the goal while avoiding all obstacles.

The proposed trajectory aligns with the user requirements by going to the goal and avoiding all obstacles except `obstacle7`. The drone is inside `obstacle7` at times [14 15 16 17].

```
<rejected>
```

Assistant:

Since the drone should avoid all obstacles, including `obstacle7`, I will update the specification to ensure that the drone avoids all obstacles completely.

The new STL specification is: `<STL_formulas.inside_cuboid(objects["goal"]).eventually(0, 35) &
STL_formulas.outside_cuboid(objects["obstacle1"]) & STL_formulas.outside_cuboid(objects["obstacle2"]) &
STL_formulas.outside_cuboid(objects["obstacle3"]) & STL_formulas.outside_cuboid(objects["obstacle4"]) &
STL_formulas.outside_cuboid(objects["obstacle5"]) & STL_formulas.outside_cuboid(objects["obstacle6"]) &
STL_formulas.outside_cuboid(objects["obstacle7"])>`

Specification checker:

The new STL specification based on the feedback is:

```
<STL_formulas.inside_cuboid(objects["goal"]).eventually(34, 35) &
STL_formulas.outside_cuboid(objects["obstacle1"]) &
STL_formulas.outside_cuboid(objects["obstacle2"]) &
STL_formulas.outside_cuboid(objects["obstacle3"]) &
STL_formulas.outside_cuboid(objects["obstacle4"]) &
STL_formulas.outside_cuboid(objects["obstacle5"]) &
STL_formulas.outside_cuboid(objects["obstacle6"]) &
STL_formulas.outside_cuboid(objects["obstacle7"]).always(14, 17)>
```

Assistant:

The feedback provided indicates that the drone should avoid `obstacle7` at all times. I will adjust the specification accordingly.

The updated STL specification is: `<STL_formulas.inside_cuboid(objects["goal"]).eventually(0, 35) &
STL_formulas.outside_cuboid(objects["obstacle1"]) & STL_formulas.outside_cuboid(objects["obstacle2"]) &
STL_formulas.outside_cuboid(objects["obstacle3"]) & STL_formulas.outside_cuboid(objects["obstacle4"]) &
STL_formulas.outside_cuboid(objects["obstacle5"]) & STL_formulas.outside_cuboid(objects["obstacle6"]) &
STL_formulas.outside_cuboid(objects["obstacle7"]).always(0, 35)>`

Fig. 10: The specification checker corrects a faulty specification. The initial specification lacks temporal functions operating on the obstacles, so the obstacles are not always avoided. The specification checker correctly identifies the issue of traversing an obstacle from the trajectory. However, the specification checker also mistakenly corrects the time window for reaching the goal, which was not specified in the task description. Eventually, the planner generates a specification, resulting in a correct trajectory using the feedback.

B. Separate specifications

Suppose it is infeasible to generate waypoints due to an ill-defined specification. In that case, the current system cannot find the conflict in the specification if the syntax is correct. A topic worth investigating is separating the specification into parts. A method is presented to separate STL specifications in a study by Zhang et al. [36]. When the specification is separated, the LLM can analyze and reason about the parts to find these conflicts.

C. LLM fine-tuning

Previous research [11] investigated fine-tuning pre-trained LLMs to increase output specificity toward the method and syntax used. It is hard to obtain large enough datasets to perform such fine-tuning, and current methods have to do with weak supervision using few-shot learning. However, fine-tuning would be worth investigating in the future.

D. Additional suggestions for future research

Additional future research suggestions include extending the framework to apply to situations with multiple drones. An important factor when dealing with multiple drones is to avoid mutual collisions. A potential method is to generate individual specifications and trajectories, and when a collision is detected, one of the specifications is altered to include a minimum mutual distance. Luo et al. [37] propose an LTL-based framework for task allocation, which can form a basis for future research.

The system can also be used with real drones. If the scene is accurately modeled, the generated waypoints can be followed using a PID controller, as per the simulation.

Finally, multimodal LLMs have become more popular and capable during the execution of this thesis. These models can prove helpful in task planning while simultaneously interpreting a visualization of the scene. Integrating multimodal LLMs could help interpret geometrical relations between objects, like distances and situations where regions enclose one another. The literature mentioned in Section I can provide a basis for using visual and multimodal models in future research.

VI. CONCLUSION

VernaCopter, introduced in this study, allows users to control drones using natural language. An LLM translates task specifications into STL as an intermediate representation, separating task and motion planning. Syntax and semantics checkers are deployed to correct STL specifications.

Experimental studies show that the system can effectively perform simple reach-avoid tasks without needing specification correction, and it can usually successfully perform more complex and temporally dependent tasks. Furthermore, including the user in conversation increases

performance since the user can correct the LLM if necessary. Utilizing a more capable LLM in more complex scenarios considerably increases performance. The checkers provide no considerable advantage due to frequent miscorrections of correct specifications.

Overall, VernaCopter demonstrates that LLMs can successfully be applied to control drones using natural language. LLMs can effectively interpret and reason about formal language as representations of tasks when prompted appropriately. Future work can enhance the interpreting capabilities of the LLM by applying dynamic-less semantics checking, separating specifications, and fine-tuning the LLM. Furthermore, the real-world grounding of the system can be enhanced using multimodal LLMs.

VII. ACKNOWLEDGMENTS

I would like to express my gratitude to my thesis supervisors, Dr. Zhiyong Sun, Dr. Ir. Sofie Haesaert, Dr. Ing. Zengjie Zhang, and Shuhao Qi, for their guidance during the project. They have shown great enthusiasm for the subject and dedication to achieving excellent results, always in a friendly and cooperative manner.

Special thanks to all my fellow students from AI&ES for making this thesis period enjoyable and for their help throughout the project, and, particularly, Pim van den Akker, for generating Figure 1.

REFERENCES

- [1] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].
- [2] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2023. arXiv: 2312.11805 [cs.CL].
- [3] Humza Naveed et al. *A Comprehensive Overview of Large Language Models*. 2024. arXiv: 2307.06435 [cs.CL].
- [4] Sai Venkrala et al. *ChatGPT for Robotics: Design Principles and Model Abilities*. 2023. arXiv: 2306.17582 [cs.AI].
- [5] Yongchao Chen et al. *AutoTAMP: Autoregressive Task and Motion Planning with LLMs as Translators and Checkers*. 2024. arXiv: 2306.06531 [cs.RO].
- [6] Zhigen Zhao et al. *A Survey of Optimization-based Task and Motion Planning: From Classical To Learning Approaches*. 2024. arXiv: 2404.02817.
- [7] N. Wake et al. “ChatGPT empowered long-step robot control in various environments: A case application”. In: *arXiv preprint arXiv:2304.03893* (2023).

- [8] K. Valmeekam et al. “Large language models still can’t plan (a benchmark for LLMs on planning and reasoning about change)”. In: *arXiv preprint arXiv:2206.10498* (2022).
- [9] I. Singh et al. “ProgPrompt: Generating situated robot task plans using large language models”. In: *International Conference on Robotics and Automation (ICRA)*. Available: <https://arxiv.org/abs/2209.11302>. 2023.
- [10] Y. Ding et al. “Task and motion planning with large language models for object rearrangement”. In: *arXiv preprint arXiv:2303.06247* (2023).
- [11] Yixiang Jin et al. “RobotGPT: Robot Manipulation Learning From ChatGPT”. In: *IEEE Robotics and Automation Letters* 9.3 (2024), pp. 2543–2550. DOI: 10.1109/LRA.2024.3357432.
- [12] Krishan Rana et al. “SayPlan: Grounding Large Language Models using 3D Scene Graphs for Scalable Task Planning”. In: *7th Annual Conference on Robot Learning*. 2023. URL: <https://openreview.net/forum?id=wMpOMO0Ss7a>.
- [13] Wenlong Huang et al. *Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents*. 2022. arXiv: 2201.07207 [cs.LG].
- [14] M. Ahn et al. “Do as I can, not as I say: Grounding language in robotic affordances”. In: *arXiv preprint arXiv:2204.01691* (2022).
- [15] K. Lin et al. “Text2motion: From natural language instructions to feasible plans”. In: *arXiv preprint arXiv:2303.12153* (2023).
- [16] J. Wu et al. “Tidybot: Personalized robot assistance with large language models”. In: *arXiv preprint arXiv:2305.05658* (2023).
- [17] W. Huang et al. “Inner monologue: Embodied reasoning through planning with language models”. In: *arXiv preprint arXiv:2207.05608* (2022).
- [18] Arthur Bucker et al. “LATTE: LAnguage Trajectory TransformEr”. In: (Aug. 2022). URL: <http://arxiv.org/abs/2208.02918>.
- [19] Erion Plaku and Sertac Karaman. “Motion planning with temporal-logic specifications: Progress and challenges”. In: *AI Commun.* 29 (2015), pp. 151–162. URL: <https://api.semanticscholar.org/CorpusID:30347236>.
- [20] Yaqi Xie et al. “Translating Natural Language to Planning Goals with Large-Language Models”. In: (Feb. 2023). URL: <http://arxiv.org/abs/2302.05128>.
- [21] Bo Liu et al. “LLM+P: Empowering Large Language Models with Optimal Planning Proficiency”. In: (Apr. 2023). URL: <http://arxiv.org/abs/2304.11477>.
- [22] Igor Buzhinsky. “Formalization of natural language requirements into temporal logics: a survey”. In: *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*. Vol. 1. 2019, pp. 400–406. DOI: 10.1109/INDIN41052.2019.8972130.
- [23] Oded Maler and Dejan Nickovic. “Monitoring Temporal Properties of Continuous Signals”. In: vol. 3253. Jan. 2004, pp. 152–166. ISBN: 978-3-540-23167-7. DOI: 10.1007/978-3-540-30206-3_12.
- [24] Georgios E. Fainekos and George J. Pappas. “Robustness of temporal logic specifications for continuous-time signals”. In: *Theoretical Computer Science* 410.42 (2009), pp. 4262–4291. ISSN: 0304-3975. DOI: <https://doi.org/10.1016/j.tcs.2009.06.021>. URL: <https://www.sciencedirect.com/science/article/pii/S0304397509004149>.
- [25] Jules White et al. *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT*. 2023. arXiv: 2302.11382 [cs.SE].
- [26] Sabit Ekin. “Prompt Engineering For ChatGPT: A Quick Guide To Techniques, Tips, And Best Practices”. In: *TechRxiv* (2023). e-Prints posted on TechRxiv are preliminary reports that are not peer-reviewed. They should not be regarded as conclusive, guide clinical practice/health-related behavior, or be reported in the media as established information. DOI: 10.36227/techrxiv.22683919.v2.
- [27] Jindong Gu et al. *A Systematic Survey of Prompt Engineering on Vision-Language Foundation Models*. 2023. arXiv: 2307.12980 [cs.CV].
- [28] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL].
- [29] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL].
- [30] Federico Augugliaro, Angela P Schoellig, and Raffaello D’Andrea. “Generation of collision-free trajectories for a quadrocopter fleet: A sequential convex programming approach”. In: *2012 IEEE/RSJ international conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 1917–1922.
- [31] Calin Belta and Sadra Sadraddini. “Formal Methods for Control Synthesis: An Optimization Perspective”. In: *Annual Review of Control, Robotics, and Autonomous Systems* 2.1 (2019), pp. 115–140. DOI: 10.1146/annurev-control-053018-023717. eprint: <https://doi.org/10.1146/annurev-control-053018-023717>. URL: <https://doi.org/10.1146/annurev-control-053018-023717>.
- [32] Vince Kurtz and Hai Lin. “Mixed-Integer Programming for Signal Temporal Logic with Fewer Bi-

-
- nary Variables”. In: *IEEE Control Systems Letters* (2022).
- [33] Jacopo Panerati et al. “Learning to Fly—a Gym Environment with PyBullet Physics for Reinforcement Learning of Multi-agent Quadcopter Control”. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2021, pp. 7512–7519. DOI: 10.1109/IROS51168.2021.9635857.
- [34] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.
- [35] Vince Kurtz and Hai Lin. “A More Scalable Mixed-Integer Encoding for Metric Temporal Logic”. In: *IEEE Control Systems Letters* 6 (2022), pp. 1718–1723. DOI: 10.1109/LCSYS.2021.3132839.
- [36] Zengjie Zhang and Sofie Haesaert. “Modularized Control Synthesis for Complex Signal Temporal Logic Specifications”. In: *2023 62nd IEEE Conference on Decision and Control (CDC)*. 2023, pp. 7856–7861. DOI: 10.1109/CDC49753.2023.10383263.
- [37] Xusheng Luo and Michael M. Zavlanos. “Temporal Logic Task Allocation in Heterogeneous Multirobot Systems”. In: *IEEE Transactions on Robotics* 38.6 (2022), pp. 3602–3621. DOI: 10.1109/TRO.2022.3181948.