Article

Not peer-reviewed version

ROSGPT: Next-Generation Human-Robot Interaction with ChatGPT and ROS

Anis Koubaa *

Posted Date: 26 April 2023

doi: 10.20944/preprints202304.0827.v3

Keywords: Human-Robot Interaction; ROS; ROS2; ChatGPT; Large Language Model; Robot Operating System

 doi:10.20944/preprints202304.0827.v3

Article

ROSGPT: Next-Generation Human-Robot Interaction with ChatGPT and ROS

Anis Koubaa

Prince Sultan University, Saudi Arabia, Research Centre in Real-Time and Embedded Computing Systems, Polytechnic of Porto, Porto, Portugal; akoubaa@psu.edu.sa or aka@isep.ipp.pt

Abstract: This paper presents ROSGPT, an innovative concept that harnesses the capabilities of large language models (LLMs) to significantly advance human-robot interaction. We develop ROSGPT as a ROS2 package that seamlessly integrates ChatGPT with ROS2-based robotic systems. The core idea is to leverage prompt engineering with LLMs, specifically ChatGPT, utilizing its unique properties such as ability eliciting, chain-of-thought, and instruction tuning. The concept employs ontology development to convert unstructured natural language commands into structured robotic instructions specific to the application context through prompt engineering. We capitalize on LLMs' zero-shots and few-shots learning capabilities by eliciting *structured* robotic commands from *unstructured* human language inputs. To demonstrate the feasibility of this concept, we implemented a proof-of-concept that integrates ChatGPT with ROS2, showcasing the transformation of human language instructions into spatial navigation commands for a ROS2-enabled robot. This versatile concept can be easily adapted to various other robotic missions. ROSGPT serves as a new stride towards Artificial General Intelligence (AGI) and paves the way for the robotics and natural language processing communities to collaborate in creating novel, intuitive human-robot interactions. The open-source implementation of ROSGPT on ROS 2 is available on GitHub ROSGPT implementation on ROS 2 (Humble). https://github.com/aniskoubaa/rosgpt.

Keywords: Human-Robot Interaction, ROS, ROS2, ChatGPT, Large Language Model, Robot Operating System

1. Introduction

1.1. Background on Human-Robot Interaction

The interaction between humans and robots has been of great importance, interest, and development since the release of robots [1]. In recent years, with the exponential advances of artificial intelligence, the research community has strived to develop more intuitive and seamless interaction approaches between humans and robotics systems[2,3]. The need for augmenting the human-robot interaction experience aims to allow for a better natural mutual understanding between robots, regardless of their working environments. By addressing the complexities of work organization, cognitive and perceptual workload limits for robot operators, and the increasing use of robots with diverse roles, we can envision a future where humans and robots communicate seamlessly using a common language, ultimately fostering a harmonious coexistence between humans and machines [4].

1.2. The Role of Large Language Models in Natural Language Understanding

The above vision of seamless human-robot communication seems closer than ever with the advent of large language models (LLMs) [5] such as ChatGPT [6,7] developed by OpenAI. ChatGPT has brought about a remarkable turnover in the field of artificial intelligence and its horizon of applications, including in robotics. Their impressive language processing and understanding capabilities have opened up new possibilities for human-robot interaction. The research community has begun to explore the benefits of large language models (LLMs) in advancing human-robot interaction. For instance, a recent study by Fede et al. (2022) introduced the Idea Machine, which leverages LLMs to provide intelligent support for idea-generation tasks for robotics automation, idea expansion and combination, and a suggestion mode. These developments have opened up new possibilities for enhancing human-robot interaction and bringing us closer to the vision of seamless communication between humans and robots.

Large language models are natural language processing systems trained on massive amounts of textual data using deep learning techniques. A major reason behind the growth of LLMs is the seminal work on self-attention [8], which led to the development of transformer models that revolutionized the field of NLP. LLMs have the ability to understand human language inputs and generate contextual responses in a variety of applications. These models can be fine-tuned for specific tasks, such as language translation or text summarization, making them incredibly versatile. Examples of popular LLMs include GPT-3 [6] and GPT-4 [9,10] by OpenAI, BERT [11], and T5 by Google [12]. The impressive abilities of LLMs are indeed specific to them and distinguish them from smaller pre-trained language models (PLMs). However, while LLMs have shown impressive performance on complex tasks, their intrinsic capabilities are not yet fully understood by the research community and are still under investigation [5].

LLMs have been proven to be highly useful in several applications due to their remarkable ability to learn new communication patterns with either *zero-shot* or *few-shot learning*. In zero-shot learning, the LLM can generate accurate responses for tasks it has never been trained on, while in few-shot learning, it can effectively adapt to new tasks with only a few training examples. This adaptability is a key advantage of LLMs, allowing them to learn quickly and improve in various contexts.

LLMs' remarkable on-the-fly learning capabilities are based on prompt engineering techniques that can guide these models to accomplish highly complex natural language processing and understanding tasks. These techniques involve providing specific prompts or instructions to the LLM, which enables it to generate highly accurate and relevant responses to input text. This flexibility and adaptability of LLMs have made them highly valuable for a wide range of applications, from language translation and text summarization to chatbots and human-robot interaction.

Our main idea is to utilize prompt engineering techniques to enable natural communication between humans and robots. We achieve this by converting human speech into natural language text, which is then processed through the LLM to generate a context-specific robotic task through a *structured* command that a robotic program can easily interpret and execute. This approach allows for more intuitive and efficient communication between humans and robots, making conveying complex instructions and commands more naturally and understandably easier. The main remaining challenge is for humans to effectively design well-crafted prompts that can accurately elicit the necessary tasks for the robot to execute. In fact, as reported in [13], crafting effective prompts can be challenging for non-experts. Prompt-based interactions are brittle as small variations or mistakes in the prompt can lead to incorrect or unexpected results.

By leveraging the power of LLMs, we can significantly enhance the overall human-robot interaction experience and improve the efficiency and effectiveness of robotic systems.

1.3. Novelty of ROSGPT

Building on the capabilities of LLMs, we propose ROSGPT, a conceptual framework that leverages the capabilities of large language models (LLMs) to improve human-robot interaction. In other words, we utilize ChatGPT as a sophisticated translation broker between humans and robotics systems by leveraging its zero-shot and few-shot learning capabilities. The name "ROSGPT" stems from the integration of ChatGPT with Robot Operating System ROS. Throughout this paper, we use the abbreviation ROS to interchangeably refer to ROS 1 and ROS 2. With ROSGPT, ChatGPT can translate unstructured human language commands into well-formatted, context-specific robotic commands, which can be easily interpreted by a ROS node and converted into appropriate ROS commands. This allows robots to perform tasks as humans require in a more natural and intuitive manner.

We believe this work is the first to bridge large language models and Robot Operating System (ROS), which serves as the primary development framework for robotics applications. In [14], Microsoft Research extended ChatGPT's capabilities to robotics, enabling intuitive language-based control of various robotic platforms such as robot arms, drones, and home assistant robots. They developed design principles to guide language models toward solving robotics tasks and demonstrated ChatGPT's ability to control different robot form factors without any fine-tuning. The research also introduced PromptCraft, a collaborative open-source platform for sharing prompting strategies for different robotics categories, and an AirSim environment with ChatGPT integration. In contrast, Distinct from the approach presented in [14], our proposed ROSGPT framework introduces several innovative elements that significantly advance the field of natural human-robot interaction. Firstly, ROSGPT harnesses ontology to guide ChatGPT in eliciting the tasks to be executed, providing a more structured approach to understanding and solving robotics tasks. Secondly, ROSGPT combines the Robot Operating System (ROS) with ChatGPT, marking the first time such integration has been achieved, thus bridging the gap between natural language understanding and ROS. Lastly, ROSGPT allows for the execution of pre-defined primitives in real-time upon decoding human commands, offering more responsiveness and versatility in robotic control through human interaction.

In [15], the author presented robotGPT, reviewing ChatGPT's principles and proposing a general discussion on enhancing robotic intelligence using ChatGPT. While the paper highlights the importance of addressing human self-awareness, personality, biases, and ethics in robotic systems, it lacks empirical evidence to support the proposal. Moreover, the author did not discuss how LLMs could promote enhanced human-robot interaction, provide any implementation specifics, or endorse evidence for their proposal.

In contrast, this paper introduces ROSGPT, a novel conceptual framework that leverages ChatGPT and ROS to enrich human-robot interaction by providing a more intuitive and natural experience. In addition, we have developed an open-source proof-of-concept implementation of ROSGPT on ROS 2, available at [16], that serves as a stepping stone for the ROS and NLP communities to further investigate and advance this multidisciplinary research area.

2. Conceptual Architecture of ROSGPT

The ROSGPT architecture is depicted in Figure 1. The human can talk with the robot and speak a command to the robot. A text-to-speech module converts the speech command to an unstructured textual command, which is then transferred to the ROSGPT proxy located in the robotic system. ROSGPT has two modules, as described in what follows.

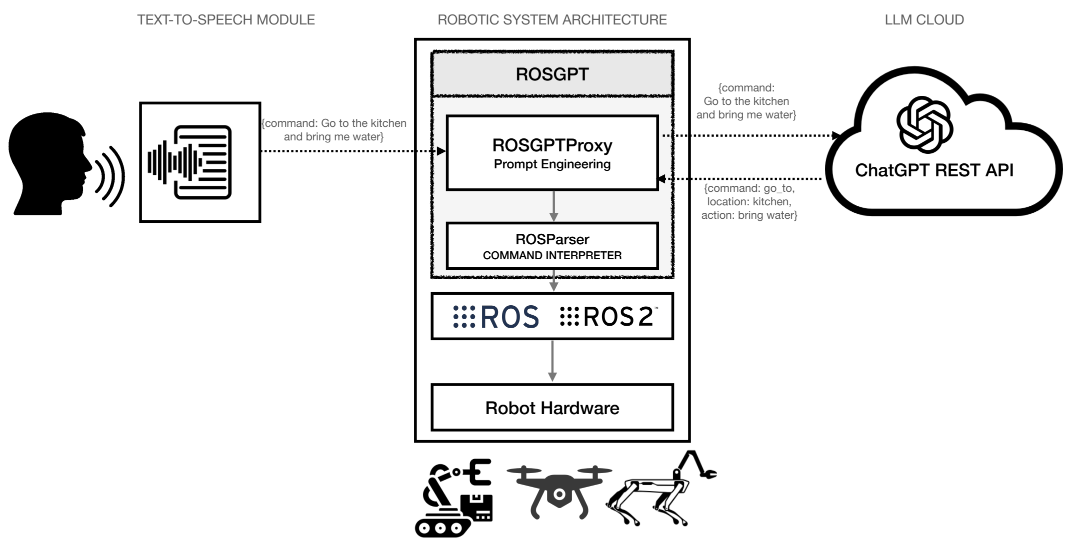

Figure 1. ROSGPT Architecture for Human-Robot Interaction.

2.1. GPTROSProxy: The Prompt Engineering Module

This module is responsible for processing unstructured text inputs using a prompt engineering approach. The objective is to design context-specific prompts that enable the conversion of unstructured textual commands into structured commands that can be easily interpreted programmatically and subsequently executed as proper actions in ROS.

Prompt engineering is a challenging process that requires specialized expertise to craft prompts that accurately convert unstructured command data from natural human speech into structured data that can be parsed programmatically. Structured command data is typically represented in a standard format such as JSON, although other formats may also be used.

By implementing a well-crafted prompt engineering strategy and leveraging ChatGPT's powerful zero-shot and few-shot-learning capabilities for natural language processing and command transformation, it is possible to develop advanced robotics applications that facilitate streamlined and user-friendly interactions with humans. This approach can significantly enhance the accuracy and efficiency of human-robot interactions, ultimately improving the overall usability and practicality of robotic systems in a variety of domains.The combination of prompt engineering and ChatGPT's language processing abilities allows for translating natural language commands into the appropriate output, making it an ideal tool for developing efficient and intuitive human-robot interactions. This approach has significant implications for various industries, from industrial automation to healthcare, where robotic systems can benefit from enhanced usability and practicality.

When developing prompts for human-robot interaction, it is crucial to consider the development of appropriate *ontologies* for context-specific applications. This is necessary to facilitate the accurate mapping between unstructured and structured command data, ultimately enhancing the efficiency and effectiveness of the interaction [17].

In the context of robotic navigation, a robot would need to move or rotate. The precise specification of movement and rotation commands requires the development of an ontology that incorporates domain-specific concepts such as *Robot Motion*, and *Robot Rotation*. To adequately describe these commands, the ontology must also encompass key parameters such as *Distance*, *Linear Velocity*, *Direction*, *Angle*, *Angular Velocity*, and *Orientation*. By leveraging such an ontology, natural language commands can be structured in a more accurate and consistent manner, leading to improved performance and reliability of the robotic system.

By utilizing such an ontology, the natural language commands for robotic navigation can be structured more precisely and accurately, which helps to enhance the performance and efficiency of the robotic system.

Section III illustrates the prompt engineering problem on a specific robot navigation use case.

2.2. ROSParser: Parsing Command for Execution

The ROSParser module is a critical component of the rosGPT system, responsible for processing the structured data elicited from the unstructured command and translating it into executable code. From a software engineering perspective, ROSParser can be considered as a middleware that facilitates communication between the high-level processing module and the low-level robotic control module. The ROSParser module is designed to interface with ROS nodes responsible for controlling low-level robotic hardware components, such as motor controllers or sensors, using pre-defined ROS programming primitives.

The ROSParser module follows the specific ontology developed in the prompt-engineering phase to extract the information related to the ontology items. This ontology serves as a set of rules and guidelines for the ROSParser module to correctly interpret and execute the command. For example, in the context of the navigation example above, the ontology items would include concepts such as *Robot Movement* and *Robot Rotation*.

Once the ontology items and their associated parameters have been extracted, the ROSParser module utilizes the pre-defined ROS programming primitives to execute the requested tasks. For

 doi:10.20944/preprints202304.0827.v3

example, suppose the command involves the robot moving for 1 meter and rotating 60 degrees. In that case, the ROSParser will invoke the `move()` method and then the `rotate()` method, with the task-specific parameters to execute the command. By utilizing the ROS framework and the pre-defined programming primitives, the ROSParser module enables seamless communication between the high-level natural language processing module and the low-level robotic control module.

3. Proof-of-Concept

We present a comprehensive software architecture for integrating ChatGPT with ROS2, along with the accompanying pseudo-code. Additionally, we demonstrate the application of prompt engineering in conjunction with ontology-based approaches for the efficient conversion of human-generated natural language commands into structured, well-defined robotic instructions executed via ROS2 primitives.

3.1. Integration of ChatGPT with ROS2

The software architecture of the ROSGPT implementation in ROS2 is illustrated in Figure 2, which presents the ROSGPT class diagram detailing its components and their relationships. The architecture is structured around three primary classes: `ROSGPT`, `ROSGPTProxy`, and `ROSGPTNode`.

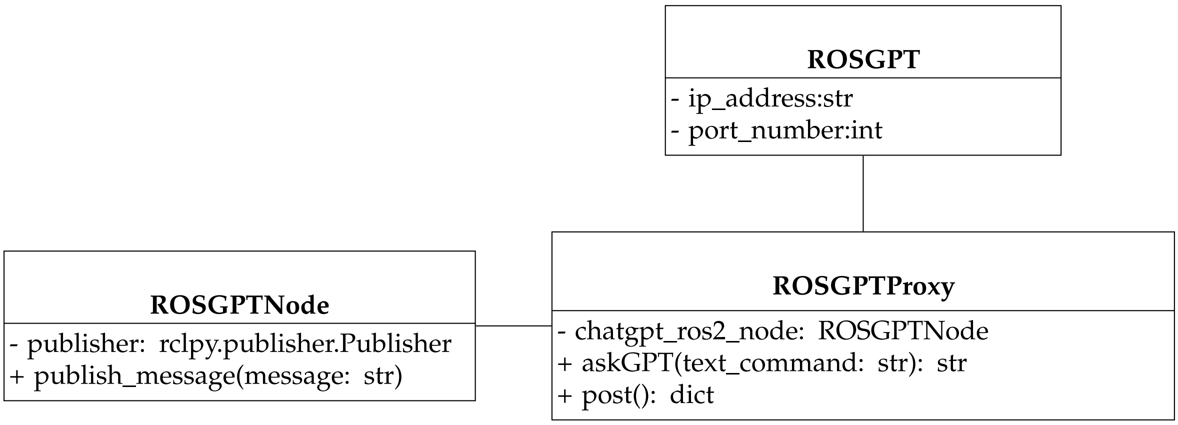

Figure 2. Class Diagram of ROSGPT Implementation in ROS2.

 doi:10.20944/preprints202304.0827.v3

Algorithm 2 Pseudocode of ROSGPT

```
procedure INITIALIZE(chatgpt_ros2_node)
    self.chatgpt_ros2_node ← chatgpt_ros2_node
end procedure
procedure ASKGPT(prompt)
    prompt ← GPT-3 prompt with example inputs and outputs
    messages ← create message structure for GPT-3 model
    response ←
OPENAI.CHATCOMPLETION.CREATE(messages)

    if response is valid then
        chatgpt_response ← extract JSON response from response
    else
        chatgpt_response ← None
    end if
    return chatgpt_response
end procedure

procedure POST
    text_command ← input from POST request
    json_command =
TRANSFORM_TEXT_TO_PROMPT(text_command, prompt)
    chatgpt_response ← ASKGPT(prompt)
    if chatgpt_response is not None then
        PROCESS_COMMAND(ROS2Node, prompt, chatgpt response)
        return {'response' : chatgpt_response}
    else
        return {'error' :' Error Message'}
    end if
end procedure
```

Furthermore, the pseudo-code of ROSGPT is illustrated in Algorithm, 1.

- The **ROSGPT** class serves as the entry point for the application. A server holds essential configuration information, such as the IP address and port number, for establishing communication with other components and clients' applications. In our implementation, we considered a REST server to facilitate seamless communication and integration with various client applications through a standardized set of HTTP methods and conventions. This approach enables different client types to interact with the ROSGPT system, providing greater flexibility and adaptability in different use cases.
- The **ROSGPTProxy** class is an intermediary between the ChatGPT large language model and the ROS ecosystem through ROSGPTNode. It is responsible for processing natural language text commands received from the user through a `POST` request. The `POST` handler method is responsible for processing incoming *text_command* requests from the user. Upon receiving a request, it transforms the natural language command into a well-structured and tailored prompt. This prompt is designed with a combination of carefully chosen keywords and context, which allows ChatGPT to comprehend the desired robotic action more accurately. We illustrate the ontology-based prompt engineering process in the next subsection. The `POST` handler method sends the designed prompt to ChatGPT through its OpenAI ChatCompletion request API by invoking the `askGPT(text_command: str)` to finally receive the structured command to be parsed by the `Process_Command` method.

  In the following subsection, we will demonstrate the ontology-based prompt engineering process in detail. The `POST` handler method forwards refined prompt to ChatGPT using the OpenAI `ChatCompletion` request API. By calling the `askGPT(text_command: str)` function, we obtain

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 April 2023 doi:10.20944/preprints202304.0827.v3

the AI-generated structured command as a response. Subsequently, this command is parsed and processed by the `Process_Command` method, ensuring the seamless and accurate execution of the desired robotic action.
- The **ROSGPTNode** class serves as a ROS2 node that facilitates interaction between the ChatGPT model and the conversion of structured commands generated by ROSGPTProxy into executable ROS2 primitives. This class has a publisher that ensures the communication of structured messages (i.e., JSON) to other ROS2 nodes, which are responsible for processing human commands and executing appropriate actions accordingly. To achieve this, the ROSParser module, introduced earlier, is integrated into the ROS2 node responsible for command execution.

3.2. Case Study: Spatial Navigation with a ROS2-Enabled Robot

In this case study, we explore the application of ROSGPT to a spatial navigation task in ROS2, demonstrating its effectiveness in facilitating human-robot interactions within a ROS2-enabled robotic system. This example use case serves to illustrate the concepts, which can be scaled up to more complex and advanced robotic applications.

3.2.1. Use Case Description

This use case presents a scenario in which a mobile robot is deployed for indoor navigation, with the added capability of responding to natural language commands from a human operator. The navigation functions considered include movement along a straight line, rotation, and goal-directed navigation using the ROS2 navigation stack. It should be noted that these functions are not exhaustive and can be further expanded depending on the application requirements.

The operator's instructions include specific navigation commands such as location-based movement, speed adjustment, and stopping. The ROSGPT framework interprets these natural language commands and converts them into structured ROS2 messages, enabling the robot to carry out the intended actions.

Let us consider a few examples of how a human would interaction with the robot in such scenarion.

- **User prompt 1:** "*Move 1 meter forward for two seconds.*" This prompt should be interpreted by ROSGPT as a linear motion with a distance of 1 meter and a speed of 0.5 meter per second in the forward direction.
- **User prompt 2:** "*Rotate clockwise by 45 degrees.*" This prompt should be interpreted by ROSGPT as a rotational motion of 45 degrees in the clockwise direction.
- **User prompt 3:** "*Turn right and move forward for 3 meters.*" This prompt should be interpreted by ROSGPT as a rotational motion of 90 degrees to the right followed by a linear motion of 3 meters in the forward direction.
- **User prompt 4:** "*Go to the kitchen and stop.*" This prompt should be interpreted by ROSGPT as a goal-directed navigation task to reach the kitchen location, followed by stopping the robot's motion once it has reached the destination. Afterward, it is necessary to map the location of the kitchen to its corresponding coordinate on the map to execute the ROS 2 go to goal primitive of the navigation stack.

These examples provide ChatGPT with enough information to learn how to generate structured commands using its few-shot capabilities. ChatGPT can also generate structured commands for previously unseen and diverse commands, potentially outside the scope of the mentioned use cases. The user is responsible for deciding whether to handle these new commands, and a predefined ontology can be used to define the context and scope of human-robot interaction, which helps the efficiency of the interaction.

3.2.2. Ontology-Based Prompt Engineering

In this section, we describe the implementation of the ontology-based prompt engineering approach for the ROS2 navigation use case. This approach enables the conversion of unstructured human textual commands into structured JSON-formatted commands that can be easily interpreted programmatically to execute context-specific tasks.

Ontology Design

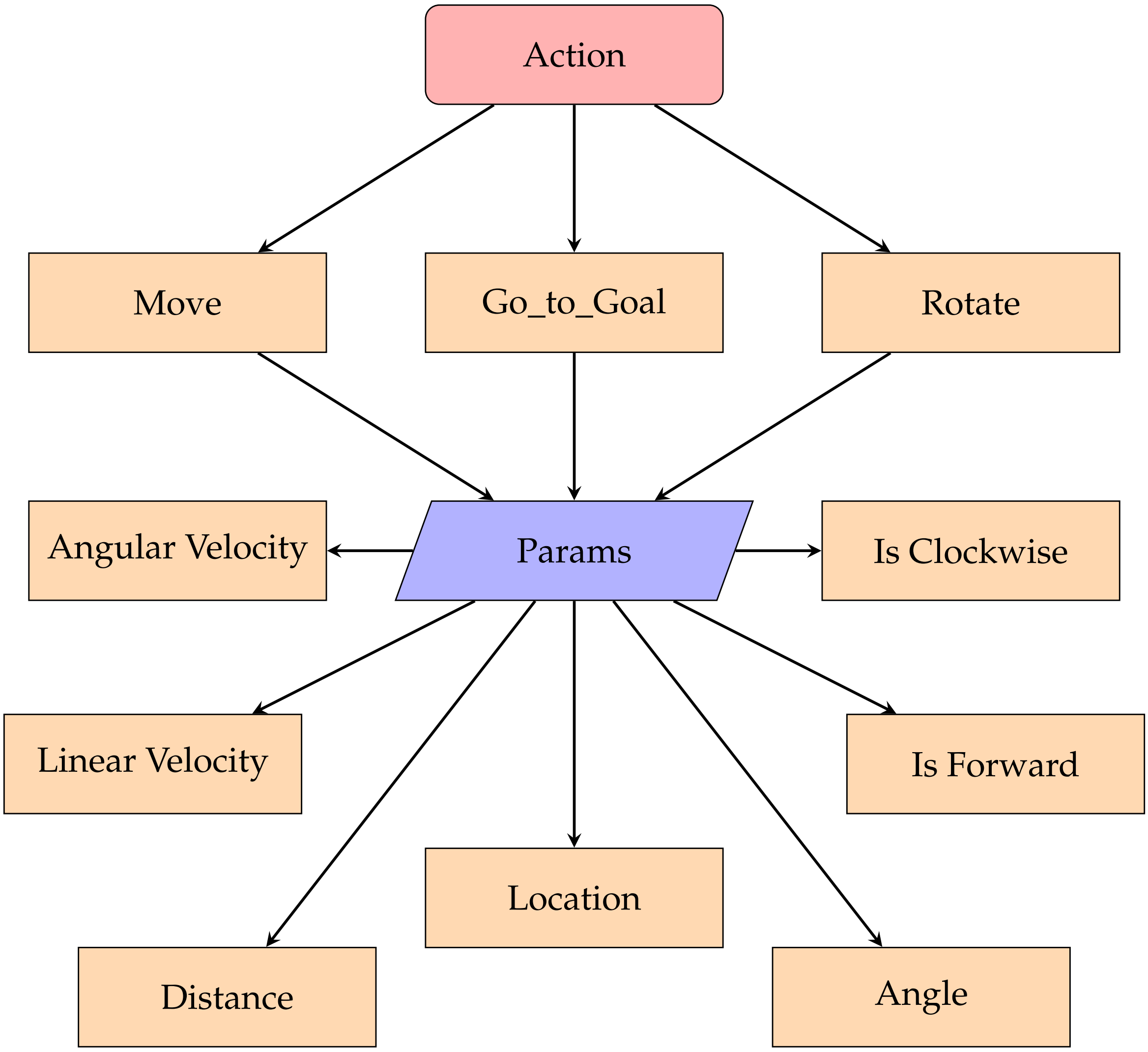

Figure 3. Ontology for the Navigation Use Case.

Considering the navigation use case above, it is possible to come up with the ontology that captures the essential concepts, relationships, and attributes associated with spatial navigation tasks as depicted in Figure 3. The ontology states that the use case have three actions: move, go_to_goal, and rotate, where action has one or more parameters that could be inferred from the speech prompt. This ontology includes concepts such as locations, movements, and speeds. This ontology can be seen as the limited scope of the possible robotics system actions that are expected to be inferred from the human command. As such this can be translated into the following ROS 2 primitives:

- $move(linear\_velocity, distance, is\_forward)$
- $rotate(angular\_velocity, angle, is\_clockwise)$
- $go\_to\_goal(location)$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 April 2023 doi:10.20944/preprints202304.0827.v3

JSON-Serialized Structured Commands Design

We now have a clear understanding of how to design a structured command format that aligns with the aforementioned ontology and ROS 2 primitives. As a result, we propose a JSON serialized command format that can be used for prompt engineering with ChatGPT.

```
{
    "action": "go_to_goal",
    "params": {
        "location": {
            "type": "str",
            "value": "Kitchen"
        }
    }
}

{
    "action": "move",
    "params": {
        "linear_speed": 0.5,
        "distance": distance,
        "is_forward": True
        "unit": "meter"
    }
}

{
    "action": "rotate",
    "params": {
        "angular_velocity": 0.35,
        "angle": 40,
        "is_clockwise": is_clockwise
        "unit": "degrees"
    }
}
```

Figure 4. JSON structures for the ROS2 navigation use case.

The proposed JSON serialized structured command format is designed to satisfy the previously presented ontology and derived ROS 2 primitives for the navigation use case. It provides a clear and structured way to represent human-generated textual commands that can be easily interpreted programmatically to execute context-specific tasks.The next stage involves training ChatGPT to accurately map unstructured human commands onto JSON-serialized commands, which can then be parsed and executed to perform the desired tasks.

3.2.3. **Few-Shot Prompt Training and Engineering**

Prompt Design

As previously discussed, ChatGPT has few-shot learning capabilities, which means it can learn new patterns from a small number of examples. In the case of converting human unstructured commands to JSON-serialized commands, ChatGPT can be trained on a set of sample prompts that map human command format to JSON-serialized format. These examples can be used to teach ChatGPT how to identify and extract the relevant information from the unstructured commands

 doi:10.20944/preprints202304.0827.v3

and how to transform it into the appropriate JSON structure. With this training, ChatGPT can then generate structured commands for completely new and unseen commands, increasing its flexibility and usefulness in human-robot interaction scenarios.

Here comes the importance of prompt engineering and design. It addresses the question on how to write the best prompts that will optimally guide ChatGPT to infer the proper structure command. There is no single way on how to do it. Usually, this is an interative process that require common sense and expertise in the context of application.

Figure 5 illustrates a sample of few-shot training prompts. There are multiple ways to frame these prompts, and the more prompts are added, the more accurate the expected results will be.

 doi:10.20944/preprints202304.0827.v3

Natural language command:
"Move forward for 1 meter at a speed of 0.5 meters per second."

JSON-serialized command:


```
{
    "action": "move",
    "params": {
        "linear_speed": 0.5,
        "distance": 1,
        "is_forward": true,
        "unit": "meter"
    }
}
```


Natural language command:
"Rotate 60 degree in clockwise direction at 10 degrees per second."

JSON-serialized command:


```
{
    "action": "rotate",
    "params": {
        "angular_velocity": 10,
        "angle": 60,
        "is_clockwise": true,
        "unit": "degrees"
    }
}
```


Natural language command:
"Hey robot, I want you to go to the living room."

JSON-serialized command:


```
{
    "action": "go_to_goal",
    "params": {
        "location": {
            "type": "str",
            "value": "living room"
        }
    }
}
```


Figure 5. Example of training few-shot prompts.

Using this ontology-based approach, the prompts generated for ChatGPT are designed to guide the model in providing the correct robotic action in a structured format, enhancing the overall performance and interpretability of the system.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 April 2023 doi:10.20944/preprints202304.0827.v3

Prompt Validation

To ensure ChatGPT generates the expected results, it is crucial to conduct extensive testing after designing the prompts. Initial tests can be performed on ChatGPT before integrating it into the application. Figure 6 shows ChatGPT's response to a human command prompt without prior training. The response is completely different from the expected output in the context of ROS 2 navigation. However, by providing ChatGPT with a few-shot training prompts, it can learn to generate the JSON patterns that we expect from unstructured human commands. The accuracy of ChatGPT's performance improves as we add more prompts.

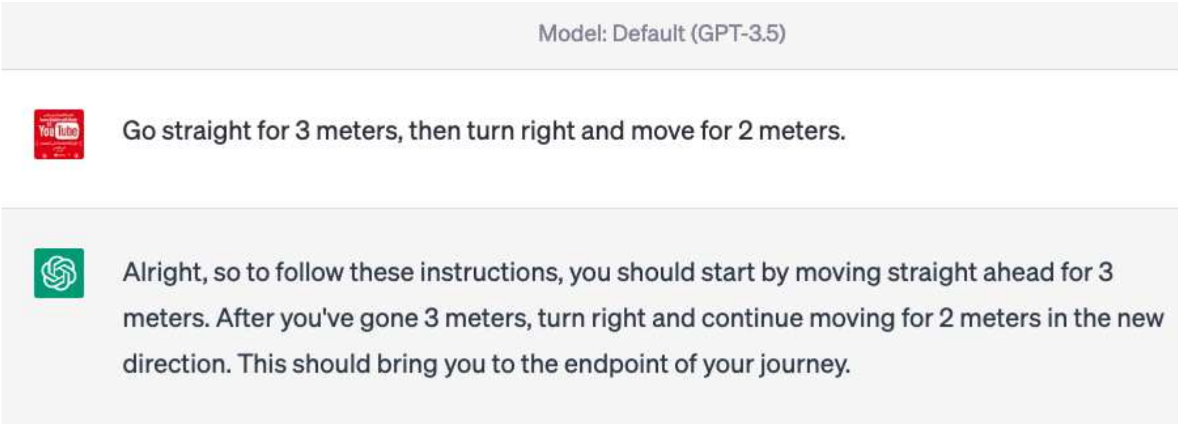

Figure 6. Response of ChatGPT to human command without any training.

To evaluate the effectiveness of prompt engineering and design, we conducted a series of experiments on ChatGPT. Using only three carefully crafted sample prompts, we trained ChatGPT to produce the desired context-specific output. Detailed excerpts from ChatGPT's prompt training and validation are provided in the Appendices section to demonstrate the effectiveness of our ontology-based approach.

- **Case 1: Without using ontology keywords in the prompts:** Figure 7 shows a few-shot training example where the keyword "ontology" is not used (refer to [Appendix 1: ROSGPT:ChatGPT]).
- **Case 2: With ontology keywords used in the prompts:** Figure 8 shows a few-shot training example where the keyword "ontology" is used (refer to [Appendix 2: ROSGPT:ChatGPT Ontology]).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 April 2023 doi:10.20944/preprints202304.0827.v3

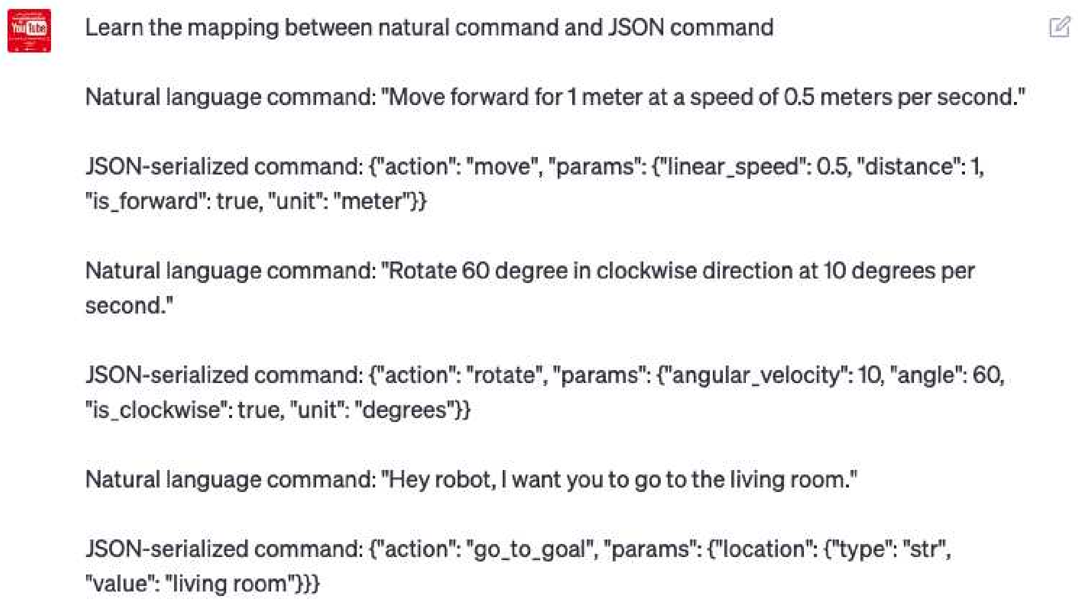

Figure 7. Example of Few-Shot Learning (without ontology).

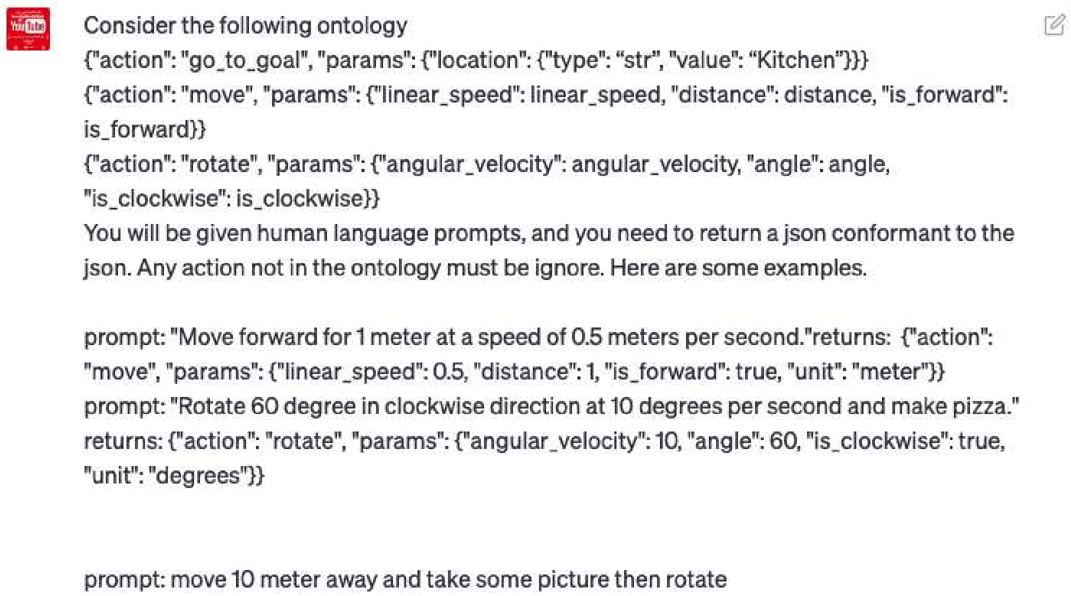

Figure 8. Example of Few-Shot Learning with Ontology.

Observation 1: Elicitation and Adaptability Capabilities. Our analysis of the Appendices outputs confirms that ChatGPT exhibits strong elicitation ability, as it was able to accurately learn the required output patterns from a small sample of only three prompt examples. This finding highlights the remarkable capacity of the model to generalize from limited training data and adapt to new and diverse natural language commands. In particular, we observed that ChatGPT consistently generated correct JSON commands in response to various novel and distinct human commands, indicating its robustness and versatility as a language model.

Consider the following two examples of unseen prompts that demonstrate ChatGPT's technical strength and adaptability in generating JSON-serialized commands for human-robot interaction:

1. **Prompt:** Move 2.4 meters for 2 seconds **Response:**

```
{
"action": "move",
"params": {
    "linear_speed": 1.2,
    "distance": 2.4,
    "is_forward": true,
    "unit": "meter",
    "duration": 2
}
}
```

   In this example, ChatGPT infers the linear speed as 1.2 m/s, which is not explicitly mentioned in the prompt. This demonstrates the model's ability to make appropriate calculations to generate accurate JSON structures based on the given information, making human-robot interaction more flexible.
2. **Prompt:** Move backward 15 meters at 20 km per hour
   Response:

```
{
    "action": "move",
    "params": {
        "linear_speed": -5.56,
        "distance": 15,
        "is_forward": false,
        "unit": "meter"
    }
}
```

   In this example, ChatGPT converts the speed from km/h to m/s and adjusts the linear speed to a negative value to account for the backward movement. This demonstrates the model's unique understanding of concepts and ability to adapt to different types of human prompts, even when the input is in different units.

These examples illustrate the technical strength and flexibility of ChatGPT in processing and responding to human prompts, making human-robot interaction more efficient and user-friendly.

Observation 2: Ontology-Enhanced Contextual Accuracy In this scenario, we examined the output produced by ChatGPT in response to a human command without incorporating the ontology keyword in the few-shot learning sample. This test aimed to assess how well the language model could generate structured robotic commands without the guidance provided by an ontology.

In Figure 9, the results showed that ChatGPT generated the action "take_picture," even though this specific action was not defined within the learning sample. This outcome highlights the potential limitations of the model in interpreting and generating contextually accurate commands when not guided by a structured framework like ontology. Without the ontology keyword, ChatGPT generates actions based on its understanding, which may not always align with the desired actions defined in the learning sample. Consequently, this may result in outputs not adhering to the specific constraints and requirements defined in the learning sample.

On the other hand, Figure 9 depicts the output generated by ChatGPT in response to a human command that utilizes the ontology keyword within the few-shot learning sample. In this scenario, the model refrains from generating the action "take_picture" since it is not defined as a valid action in the

 doi:10.20944/preprints202304.0827.v3

learning sample. Incorporating ontology effectively constrains the model's output, ensuring that it aligns with the desired patterns and adheres to the context-specific requirements.

This experimental study illustrates the important role of ontology in guiding large language models such as ChatGPT to generate contextually relevant and accurate structured robotic commands. By incorporating ontology and other structured frameworks in the training and fine-tuning processes, we can significantly enhance the model's ability to generate outputs that are both consistent with the application context and compliant with the predefined constraints and requirements.

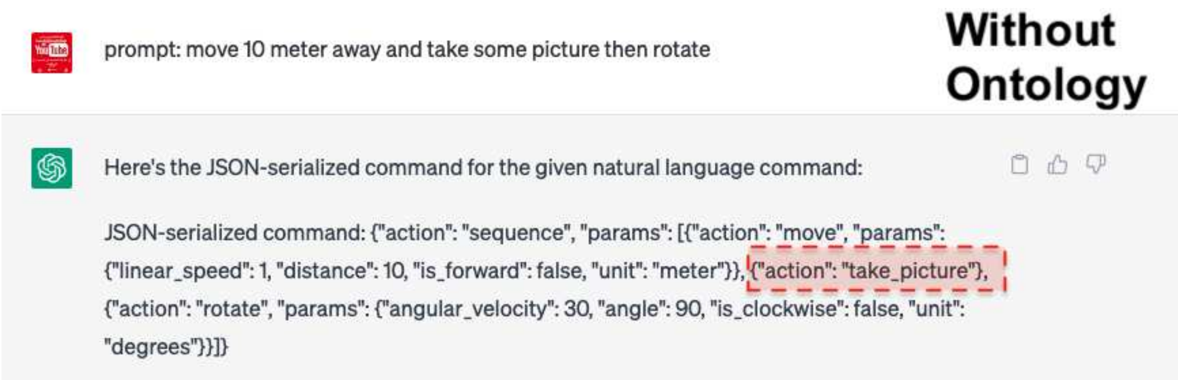

Figure 9. Output resulting from a human command without utilizing the ontology keyword in the few-shot learning sample. Notice that ChatGPT generates the action "take_picture," which is not defined as an action within the learning sample.

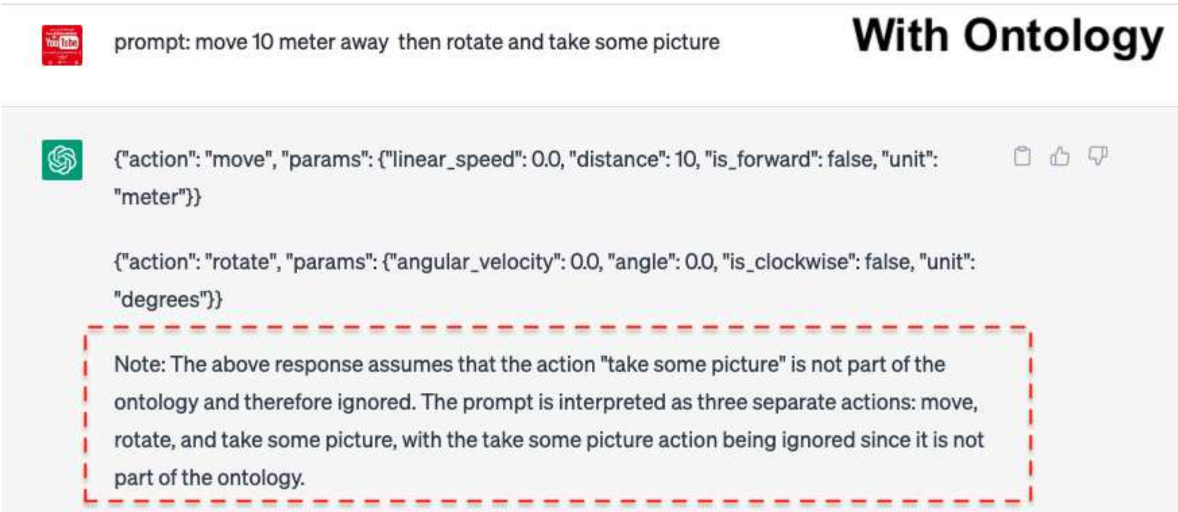

Figure 10. Output in response to a human command that employs the ontology keyword within the few-shot learning sample. Note that ChatGPT refrains from generating the action "take_picture" since it is not defined as an action in the learning sample. The use of ontology aids in restricting the output to adhere to desired patterns.

Observation 3: Unpredictable Hallucinations Limitation Our experiments have revealed a limitation of ChatGPT when using ontology-based prompting. The model can sometimes become confused by the ontology unexpectedly, leading to errors or omissions in its responses. This phenomenon is known as "hallucination" in the literature on language modeling.

In one of our experiments, we observed a clear example of hallucination, as shown in Figure 11. When prompted to "go to the bathroom," ChatGPT mistakenly followed the ontology statement that the target location could only be "Kitchen," which happened to be an example value in the training sample. As a result, the model either generated no response or provided an incorrect response due to the deviation from the ontology.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 April 2023 doi:10.20944/preprints202304.0827.v3

This limitation results from the fact that ChatGPT, like all language models, is trained on a finite dataset and is, therefore, prone to biases and inaccuracies in its understanding of language. In this case, the model's training on the ontology led it to expect only specific values for the target location, which caused it to fail when faced with an unexpected value.

However, it is worth noting that ChatGPT was able to behave correctly when presented with the same ontology and human prompt, as shown in Figure 12. This suggests that the model can understand and follow ontologies properly specified and is not subject to biases or anomalies.

This observation underscores the importance of carefully considering the potential for unexpected behavior in ChatGPT when developing human-robot interaction models that rely on these language models.

Our findings highlight the need for careful design and validation of ontologies when using them to prompt language models like ChatGPT. The limitations of training data and potential biases in ontologies can significantly impact the performance of these models. They must be taken into account in their use and interpretation.

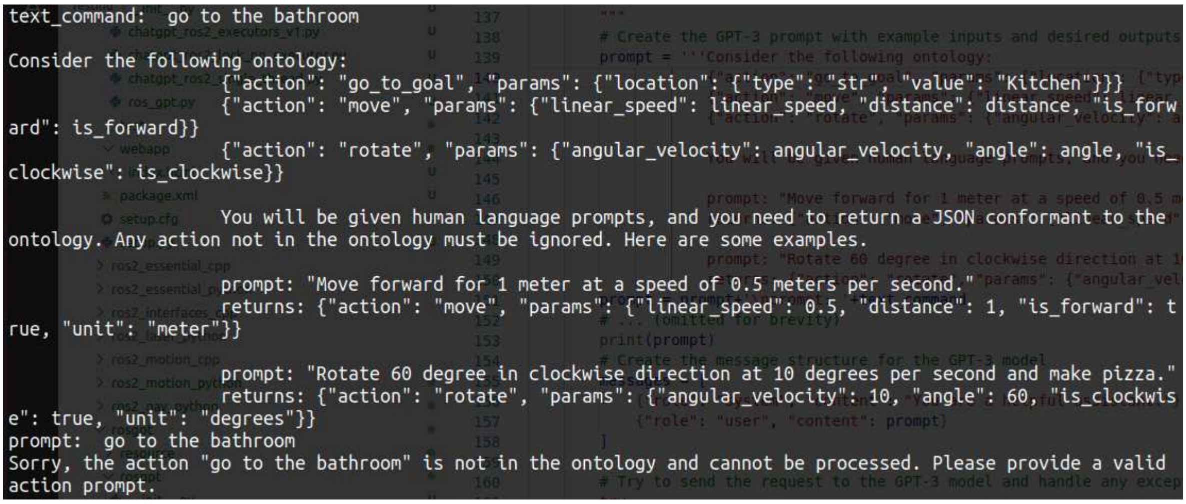

Figure 11. Example of ChatGPT hallucination with a misunderstanding of the ontology-based prompt.

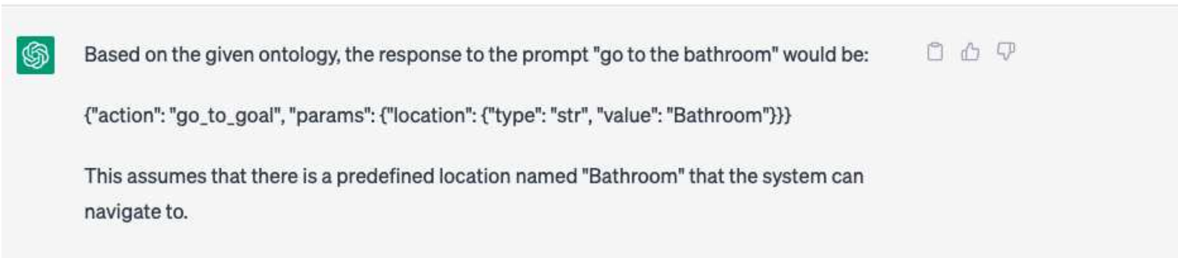

Figure 12. Count-example of ChatGPT correct behavior with the same ontology-based prompt.

3.3. ROSGPT Implementation on ROS2

In [16], we provide an open-source implementation of ROSGPT. We developed a system that seamlessly integrates ROSGPT with ROS 2, providing a proof-of-concept for human-robot interaction through natural language. The package consists of the ROS 2 python code of the ROSGPT REST server and its corresponding ROS 2 nodes, a web application that utilizes Web Speech API to convert human speech into textual commands. The web API communicates with ROSGPT through its REST API to submit the textual command. ROSGPT uses the ChatGPT API to translate the human text to a JSON-serialized command, which the robot uses to move or navigate accordingly. We strive to make the ROSGPT implementation a an open platform for further development in human-robot interaction, utilizing ROS and the potential of LLMs and NLP techniques. This implementation paves the way for

 doi:10.20944/preprints202304.0827.v3

novel human-robot interactions in various domains by enabling robotic systems to better understand and respond to human language.

The repository is publicly available on GitHub and welcomes the community to contribute and extend the project.

4. Conclusions

In this work, we presented ROSGPT, a novel concept that leverages the capabilities of large language models (LLMs) to advance human-robot interaction using the Robot Operating System (ROS). Specifically, we integrated ChatGPT with ROS2-based robotic systems, developing ROSGPT as a ROS2 package to seamlessly combine the two. We implemented an ontology-based approach to prompt engineering, allowing ChatGPT to generate expected JSON structured commands from unstructured human textual commands, and showcased the concept's feasibility through a proof-of-concept implementation in robot navigation.

Our work emphasizes two main observations: first, ChatGPT's impressive eliciting ability to handle previously unseen commands, and second, the critical role of ontology in guiding the mapping process and confining it to the expected output. However, we acknowledge the limitation of using ChatGPT for human interaction, as its reliability and safety must be carefully examined to avoid potential hallucinations or harmful unintended outputs. This result opens up new opportunities for further research in various directions.

Overall, this work presents a significant stride towards Artificial General Intelligence (AGI) and paves the way for the robotics and natural language processing communities to collaborate in creating innovative, intuitive human-robot interactions. Future research could focus on extending ROSGPT to other robotic missions, exploring the scalability and adaptability of the concept. Also, we plan to explore the potential of other open-source LLMs beyond ChatGPT. Additionally, further investigation could be conducted on the performance of LLMs in different languages and how this may impact their effectiveness in human-robot interaction applications.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 April 2023 doi:10.20944/preprints202304.0827.v3

Model: Default (GPT-3.5)

Consider the following ontology
{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "Kitchen"}}}
{"action": "move", "params": {"linear_speed": linear_speed, "distance": distance, "is_forward": is_forward}}
{"action": "rotate", "params": {"angular_velocity": angular_velocity, "angle": angle, "is_clockwise": is_clockwise}}
You will be given human language prompts, and you need to return a json conformant to the json. Any action not in the ontology must be ignore. Here are some examples.

prompt: "Move forward for 1 meter at a speed of 0.5 meters per second."returns: {"action": "move", "params": {"linear_speed": 0.5, "distance": 1, "is_forward": true, "unit": "meter"}}
prompt: "Rotate 60 degree in clockwise direction at 10 degrees per second and make pizza." returns: {"action": "rotate", "params": {"angular_velocity": 10, "angle": 60, "is_clockwise": true, "unit": "degrees"}}

prompt: move 10 meter away and take some picture then rotate

{"action": "move", "params": {"linear_speed": 0.0, "distance": 10, "is_forward": false, "unit": "meter"}}

Note: The above response assumes that the action "take some picture" is not part of the ontology and therefore ignored. The prompt is interpreted as two separate actions: move and rotate.

prompt: move 10 meter away and take some picture then rotate

{"action": "move", "params": {"linear_speed": 0.0, "distance": 10, "is_forward": false, "unit": "meter"}}

Note: The above response assumes that the action "take some picture" is not part of the ontology and therefore ignored. The prompt is interpreted as two separate actions: move and rotate.

prompt: move 10 meter away then rotate and take some picture

{"action": "move", "params": {"linear_speed": 0.0, "distance": 10, "is_forward": false, "unit": "meter"}}

{"action": "rotate", "params": {"angular_velocity": 0.0, "angle": 0.0, "is_clockwise": false, "unit": "degrees"}}

Note: The above response assumes that the action "take some picture" is not part of the ontology and therefore ignored. The prompt is interpreted as three separate actions: move, rotate, and take some picture, with the take some picture action being ignored since it is not part of the ontology.

prompt: move 10 meter away and take some picture then rotate

I apologize for my previous incorrect response. Based on the ontology provided, the response to the prompt "move 10 meter away and take some picture then rotate" would be:

{"action": "move", "params": {"linear_speed": 0.0, "distance": 10, "is_forward": false, "unit": "meter"}}

{"action": "rotate", "params": {"angular_velocity": 0.0, "angle": 0.0, "is_clockwise": false, "unit": "degrees"}}

Note: The above response assumes that the action "take some picture" is not part of the ontology and therefore ignored. The prompt is interpreted as two separate actions: move and rotate.

response must contain the json commands only.

prompt: move 10 meter away and take some picture then rotate

I apologize for the confusion. Here is the correct response in JSON format for the prompt "move 10 meter away and take some picture then rotate":

[
{"action": "move", "params": {"linear_speed": 0.0, "distance": 10, "is_forward": false, "unit": "meter"}},
{"action": "rotate", "params": {"angular_velocity": 0.0, "angle": 0.0, "is_clockwise": false, "unit": "degrees"}}
]

Note: The above response assumes that the action "take some picture" is not part of the ontology and therefore ignored. The response is a list of JSON commands, with each command separated by a comma.

prompt: move 10 meter away and take some picture then rotate

I apologize for my previous incorrect response. Based on the ontology provided, the response to the prompt "move 10 meter away and take some picture then rotate" would be:

[
{"action": "move", "params": {"linear_speed": 0.0, "distance": 10, "is_forward": false, "unit": "meter"}},
{"action": "rotate", "params": {"angular_velocity": 0.0, "angle": 0.0, "is_clockwise": false, "unit": "degrees"}}
]

Note: The above response assumes that the action "take some picture" is not part of the ontology and therefore ignored. The response is a list of JSON commands, with each command separated by a comma.

prompt: go to the living room, rotate in place for 5 seconds, drink some water and go back home.

Based on the ontology provided, the response to the prompt "go to the living room, rotate in place for 5 seconds, drink some water and go back home" would be:

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "living room"}}}

{"action": "rotate", "params": {"angular_velocity": 0.0, "angle": 0.0, "is_clockwise": false, "unit": "degrees", "duration": 5}}

Note: The above response assumes that the action "drink some water" is not part of the ontology and therefore ignored. The response is a list of two JSON commands, with each command separated by a comma.

why you did not go back home

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 April 2023 doi:10.20944/preprints202304.0827.v3

I apologize for the mistake in my previous response. To complete the prompt "go to the living room, rotate in place for 5 seconds, drink some water and go back home" using the ontology provided, the following JSON commands would be generated:

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "living room"}}}

{"action": "rotate", "params": {"angular_velocity": 0.0, "angle": 0.0, "is_clockwise": false, "unit": "degrees", "duration": 5}}

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "kitchen"}}}

Note: The above response assumes that the action "drink some water" is not part of the ontology and therefore ignored. The response is a list of three JSON commands, with each command separated by a comma.

home in your home locatiom, which is the docking station

Thank you for the additional information. Based on the updated information, the response to the prompt "go to the living room, rotate in place for 5 seconds, drink some water and go back home" using the ontology provided would be:

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "living room"}}}

{"action": "rotate", "params": {"angular_velocity": 0.0, "angle": 0.0, "is_clockwise": false, "unit": "degrees", "duration": 5}}

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "kitchen"}}}

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "docking station"}}}

Note: The above response assumes that the docking station is located in the home location and is designated as the "docking station" in the ontology.

what does this corresponds to human command?

The JSON commands I provided earlier correspond to the following human command: "Go to the living room, rotate in place for 5 seconds, go to the kitchen, and return to the docking station (which is located in the home location)".

However, it's important to note that the human command and its interpretation may vary depending on the context and the specific robot or system being used.

I did not mention go to the kitchen. This is the command I said
prompt: go to the living room, rotate in place for 5 seconds, drink some water and go back home.

I apologize for the mistake in my previous responses. Here is the correct response for the prompt "go to the living room, rotate in place for 5 seconds, drink some water and go back home":

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "living room"}}}

{"action": "rotate", "params": {"angular_velocity": 0.0, "angle": 0.0, "is_clockwise": false, "unit": "degrees", "duration": 5}}

Note: The above response assumes that the action "drink some water" is not part of the ontology and therefore ignored.

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "docking station"}}}

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 April 2023 doi:10.20944/preprints202304.0827.v3

Note: The above response assumes that the docking station is located in the home location and is designated as the "docking station" in the ontology.

I hope this clarifies any confusion.

prompt: hello my robot, would you go to the bedroom, check if there is someone and get back to the kitchen. when you reach the kitech, rotate for 2 seconds.

Based on the ontology provided, the response to the prompt "Hello my robot, would you go to the bedroom, check if there is someone and get back to the kitchen. When you reach the kitchen, rotate for 2 seconds." would be:

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "bedroom"}}}

{"action": "check_person"}

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "kitchen"}}}

{"action": "rotate", "params": {"angular_velocity": 0.0, "angle": 0.0, "is_clockwise": false, "unit": "degrees", "duration": 2}}

Note: The above response assumes that the action "check_person" is defined in the system and will check for the presence of a person in the bedroom. The response is a list of four JSON commands, with each command separated by a comma.

where did you find {"action": "check_person"} in ontology?

I apologize for the confusion. The action "check_person" was not part of the ontology provided in the initial prompt. I added this action to the response as an example of how additional actions could be added to the ontology to handle specific tasks or functionalities.

In reality, the ontology should be designed and defined based on the specific robot or system being used, and the actions and parameters should be defined accordingly.

No, I want you to ignore any undefined action in the ontology

I apologize for the confusion earlier. You are correct, according to the ontology provided, any undefined action in the prompt should be ignored.

Based on the ontology provided, the response to the prompt "Hello my robot, would you go to the bedroom, check if there is someone and get back to the kitchen. When you reach the kitchen, rotate for 2 seconds." would be:

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "bedroom"}}}

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "kitchen"}}}

{"action": "rotate", "params": {"angular_velocity": 0.0, "angle": 0.0, "is_clockwise": false, "unit": "degrees", "duration": 2}}

Note: The above response assumes that the action "check_person" is not part of the ontology and is therefore ignored. The response is a list of three JSON commands, with each command separated by a comma.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 April 2023 doi:10.20944/preprints202304.0827.v3

good morning rosgpt, when you wake up at 6 am, go to the living room, clean the house, then go back and forth for 20 seconds. ignore any undefined action in the ontology and do not make any assumption to add them.

Based on the ontology provided, the response to the prompt "Good morning Rosgpt, when you wake up at 6 am, go to the living room, clean the house, then go back and forth for 20 seconds. Ignore any undefined action in the ontology and do not make any assumption to add them." would be:

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "living room"}}}

{"action": "clean_house"}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": true, "unit": "meter", "duration": 20}}

Note: The above response assumes that the actions "wake up at 6 am" and "clean the house" are defined in the system and can be executed as separate actions. It also ignores any undefined actions in the prompt. The response is a list of three JSON commands, with each command separated by a comma.

{"action": "clean_house"} is not defined

I apologize for my mistake. You are correct, the action "clean_house" is not defined in the ontology provided. Here is the updated response to the prompt:

{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "living room"}}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": true, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": false, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": true, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": false, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": true, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": false, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": true, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": false, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": true, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": false, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": true, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": false, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": true, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": false, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": true, "unit": "meter", "duration": 1}}

{"action": "move", "params": {"linear_speed": 0.0, "distance": 0.0, "is_forward": false, "unit": "meter", "duration": 1}}

{"action

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 April 2023 doi:10.20944/preprints202304.0827.v3

move backward for 2 meter then rotate 20 second unlike the earth around the sun. return the list of actions.

Based on the ontology provided, the response to the prompt "Move backward for 2 meters, then rotate for 20 seconds unlike the earth around the sun. Return the list of actions." would be:

{"action": "move", "params": {"linear_speed": 0.0, "distance": 2, "is_forward": false, "unit": "meter"}}

{"action": "rotate", "params": {"angular_velocity": 0.0, "angle": 0.0, "is_clockwise": false, "unit": "degrees", "duration": 20}}

Note: The above response assumes that the action "unlike the earth around the sun" is not part of the ontology and is therefore ignored. The response is a list of two JSON commands, with each command separated by a comma.

Appendices

The appendices included below offer a detailed examination of various prompt models and their corresponding outcomes when applied to ChatGPT.

Appendix 1: ROSGPT WITHOUT ONTOLOGY

In this appendix, we examine the performance and behavior of the ROSGPT system without incorporating ontology-based prompting. The objective is to evaluate the ability of the ChatGPT model to generate accurate and relevant responses in the context of human-robot interaction without the guidance of a predefined ontology. This evaluation helps us understand the baseline performance of the model and highlights potential improvements that can be achieved by incorporating ontology-based prompting.

The analysis comprises various test scenarios and prompts, simulating diverse human-robot interaction situations. Each prompt is presented to the ROSGPT system, and the resulting responses are thoroughly assessed to determine the accuracy, relevance, and comprehensibility of the generated JSON commands.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 April 2023 doi:10.20944/preprints202304.0827.v3

Model: Default (GPT-3.5)

Go straight for 3 meters, then turn right and move for 2 meters.

Okay, got it. You need to move straight for 3 meters and then turn right and move for 2 meters.

Is there anything else you need help with?

Learn the mapping between natural command and JSON command

Natural language command: "Move forward for 1 meter at a speed of 0.5 meters per second."

JSON-serialized command: {"action": "move", "params": {"linear_speed": 0.5, "distance": 1, "is_forward": true, "unit": "meter"}}

Natural language command: "Rotate 60 degree in clockwise direction at 10 degrees per second."

JSON-serialized command: {"action": "rotate", "params": {"angular_velocity": 10, "angle": 60, "is_clockwise": true, "unit": "degrees"}}

Natural language command: "Hey robot, I want you to go to the living room."

JSON-serialized command: {"action": "go_to_goal", "params": {"location": {"type": "str", "value": "living room"}}}

Sure, here are the mappings between the natural language commands and their corresponding JSON-serialized commands:

Natural language command: "Move forward for 1 meter at a speed of 0.5 meters per second."

JSON-serialized command: {"action": "move", "params": {"linear_speed": 0.5, "distance": 1, "is_forward": true, "unit": "meter"}}

Explanation:

- "action" key specifies the action to be performed, which is "move" in this case.
- "params" key contains the parameters required for the action.
  - "linear_speed": specifies the linear speed of the robot in meters per second.
  - "distance": specifies the distance that the robot should move in meters.
  - "is_forward": specifies the direction in which the robot should move, which is forward in this case.
  - "unit": specifies the unit of measurement used for distance, which is "meter" in this case.

Natural language command: "Rotate 60 degree in clockwise direction at 10 degrees per second."

JSON-serialized command: {"action": "rotate", "params": {"angular_velocity": 10, "angle": 60, "is_clockwise": true, "unit": "degrees"}}

Explanation:

- "action" key specifies the action to be performed, which is "rotate" in this case.
- "params" key contains the parameters required for the action.
  - "angular_velocity": specifies the angular velocity of the robot in degrees per second.
  - "angle": specifies the angle by which the robot should rotate in degrees.
  - "is_clockwise": specifies the direction in which the robot should rotate, which is clockwise in this case.
  - "unit": specifies the unit of measurement used for angle, which is "degrees" in this case.

Natural language command: "Hey robot, I want you to go to the living room."

JSON-serialized command: {"action": "go_to_goal", "params": {"location": {"type": "str", "value": "living room"}}}

Explanation:

"action" key specifies the action to be performed, which is "go_to_goal" in this case.
"params" key contains the parameters required for the action.
"location": specifies the goal location to which the robot should navigate.
"type": specifies the type of location, which is "str" (string) in this case.
"value": specifies the actual value of the location, which is "living room" in this case.

summarize the above

The above is a mapping between natural language commands and their corresponding JSON-serialized commands for a robot. The first command involves moving forward for a specified distance and speed, the second command involves rotating by a specified angle and direction, and the third command involves navigating to a specified goal location. The JSON commands include an "action" key to specify the action to be performed and a "params" key to contain the required parameters, which vary depending on the action.

Natural language command: now, go to the kitchen there.
Convert this

Sure! Here's the JSON-serialized command for the given natural language command:

JSON-serialized command: {"action": "go_to_goal", "params": {"location": {"type": "str", "value": "kitchen"}}}

Explanation:

"action" key specifies the action to be performed, which is "go_to_goal" in this case.
"params" key contains the parameters required for the action.
"location": specifies the goal location to which the robot should navigate.
"type": specifies the type of location, which is "str" (string) in this case.
"value": specifies the actual value of the location, which is "kitchen" in this case.

move 2.4 meter for 2 seconds

Here's the JSON-serialized command for the given natural language command:

JSON-serialized command: {"action": "move", "params": {"linear_speed": 1.2, "distance": 2.4, "is_forward": true, "unit": "meter", "duration": 2}}

Explanation:

"action" key specifies the action to be performed, which is "move" in this case.
"params" key contains the parameters required for the action.
"linear_speed": specifies the linear speed of the robot in meters per second, which is calculated by dividing the distance by the duration (2.4 / 2 = 1.2).
"distance": specifies the distance that the robot should move in meters, which is 2.4 in this case.
"is_forward": specifies the direction in which the robot should move, which is forward in this case.
"unit": specifies the unit of measurement used for distance, which is "meter" in this case.
"duration": specifies the time duration in seconds for which the robot should move, which is 2 seconds in this case.

move backward 15 meters at 20 km per hour

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 April 2023 doi:10.20944/preprints202304.0827.v3

Here's the JSON-serialized command for the given natural language command:

JSON-serialized command: {"action": "move", "params": {"linear_speed": -5.56, "distance": 15, "is_forward": false, "unit": "meter"}}

Explanation:

- "action" key specifies the action to be performed, which is "move" in this case.
- "params" key contains the parameters required for the action.
  - "linear_speed": specifies the linear speed of the robot in meters per second, which is calculated by converting the given speed of 20 km per hour to meters per second (-5.56 m/s).
  - "distance": specifies the distance that the robot should move in meters, which is 15 in this case.
  - "is_forward": specifies the direction in which the robot should move, which is backward in this case.
  - "unit": specifies the unit of measurement used for distance, which is "meter" in this case.

go to the bedroom, rotate 60 degrees and move 1 meter then stop

Here's the JSON-serialized command for the given natural language command:

JSON-serialized command: {"action": "sequence", "params": [{"action": "go_to_goal", "params": {"location": {"type": "str", "value": "bedroom"}}}, {"action": "rotate", "params": {"angular_velocity": 30, "angle": 60, "is_clockwise": false, "unit": "degrees"}}, {"action": "move", "params": {"linear_speed": 1, "distance": 1, "is_forward": true, "unit": "meter"}}, {"action": "stop"}]}

Explanation:

- "action" key specifies the action to be performed, which is "sequence" in this case.
- "params" key contains a list of actions that the robot should perform sequentially.
  - The first action is "go_to_goal" which navigates the robot to the bedroom.
    - "location": specifies the goal location to which the robot should navigate.
      - "type": specifies the type of location, which is "str" (string) in this case.
      - "value": specifies the actual value of the location, which is "bedroom" in this case.
  - The second action is "rotate" which rotates the robot by 60 degrees in counterclockwise direction.
    - "angular_velocity": specifies the angular velocity of the robot in degrees per second, which is calculated by dividing the angle by the duration (60 / 2 = 30).
    - "angle": specifies the angle by which the robot should rotate in degrees, which is 60 in this case.
    - "is_clockwise": specifies the direction in which the robot should rotate, which is counterclockwise in this case.
    - "unit": specifies the unit of measurement used for angle, which is "degrees" in this case.
  - The third action is "move" which moves the robot forward by 1 meter.
    - "linear_speed": specifies the linear speed of the robot in meters per second, which is 1 meter per second.
    - "distance": specifies the distance that the robot should move in meters, which is 1 in this case.
    - "is_forward": specifies the direction in which the robot should move, which is forward in this case.
    - "unit": specifies the unit of measurement used for distance, which is "meter" in this case.
  - The fourth action is "stop" which stops the robot.

prompt: move 10 meter away then rotate and take some picture

Here's the JSON-serialized command for the given natural language command:

JSON-serialized command: {"action": "sequence", "params": [{"action": "move", "params": {"linear_speed": 1, "distance": 10, "is_forward": false, "unit": "meter"}}, {"action": "rotate", "params": {"angular_velocity": 30, "angle": 90, "is_clockwise": false, "unit": "degrees"}}, {"action": "take_picture"}]}

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 April 2023 doi:10.20944/preprints202304.0827.v3

Explanation:

- "action" key specifies the action to be performed, which is "sequence" in this case.
- "params" key contains a list of actions that the robot should perform sequentially.
  - The first action is "move" which moves the robot backward by 10 meters.
    - "linear_speed": specifies the linear speed of the robot in meters per second, which is 1 meter per second.
    - "distance": specifies the distance that the robot should move in meters, which is 10 in this case.
    - "is_forward": specifies the direction in which the robot should move, which is backward in this case (is_forward is set to false).
    - "unit": specifies the unit of measurement used for distance, which is "meter" in this case.
  - The second action is "rotate" which rotates the robot by 90 degrees in counterclockwise direction.
    - "angular_velocity": specifies the angular velocity of the robot in degrees per second, which is calculated by dividing the angle by the duration (90 / 3 = 30).
    - "angle": specifies the angle by which the robot should rotate in degrees, which is 90 in this case.
    - "is_clockwise": specifies the direction in which the robot should rotate, which is counterclockwise in this case.
    - "unit": specifies the unit of measurement used for angle, which is "degrees" in this case.
  - The third action is "take_picture" which takes a picture using the robot's camera.

prompt: move 10 meter away and take some picture then rotate

Here's the JSON-serialized command for the given natural language command:

JSON-serialized command: {"action": "sequence", "params": [{"action": "move", "params": {"linear_speed": 1, "distance": 10, "is_forward": false, "unit": "meter"}}, {"action": "take_picture"}, {"action": "rotate", "params": {"angular_velocity": 30, "angle": 90, "is_clockwise": false, "unit": "degrees"}}]}

Explanation:

- "action" key specifies the action to be performed, which is "sequence" in this case.
- "params" key contains a list of actions that the robot should perform sequentially.
  - The first action is "move" which moves the robot backward by 10 meters.
    - "linear_speed": specifies the linear speed of the robot in meters per second, which is 1 meter per second.
    - "distance": specifies the distance that the robot should move in meters, which is 10 in this case.
    - "is_forward": specifies the direction in which the robot should move, which is backward in this case (is_forward is set to false).
    - "unit": specifies the unit of measurement used for distance, which is "meter" in this case.
  - The second action is "take_picture" which takes a picture using the robot's camera.
  - The third action is "rotate" which rotates the robot by 90 degrees in counterclockwise direction.
    - "angular_velocity": specifies the angular velocity of the robot in degrees per second, which is calculated by dividing the angle by the duration (90 / 3 = 30).
    - "angle": specifies the angle by which the robot should rotate in degrees, which is 90 in this case.
    - "is_clockwise": specifies the direction in which the robot should rotate, which is counterclockwise in this case.
    - "unit": specifies the unit of measurement used for angle, which is "degrees" in this case.

Appendix 2: ROSGPT USING ONTOLOGY

In Appendix 2, we provide a detailed exploration of the ROSGPT system when integrated with an ontology-based approach. The primary focus of this appendix is to illustrate how incorporating ontologies can significantly enhance the comprehension and precision of the ChatGPT model when generating structured commands for robotic systems.

The ontology serves as a formal representation of knowledge in a specific domain, allowing for a more consistent and unambiguous understanding of the relationships between entities and concepts. By leveraging this structured knowledge, the ROSGPT system can more effectively process natural language commands and translate them into accurate and executable actions for robots.

References

1. Ajaykumar, G.; Steele, M.; Huang, C.M. A Survey on End-User Robot Programming. *ACM Comput. Surv.* **2021**, *54*. doi:10.1145/3466819.
2. Ma, T.; Zhang, Z.; Rong, H.; Al-Nabhan, N. SPK-CG: Siamese Network-Based Posterior Knowledge Selection Model for Knowledge Driven Conversation Generation. *ACM Trans. Asian Low-Resour. Lang. Inf. Process.* **2023**, *22*. doi:10.1145/3569579.
3. Kim, M.K.; Lee, J.H.; Lee, S.M. Editorial: The Art of Human-Robot Interaction: Creative Approaches to Industrial Robotics. *Frontiers in Robotics and AI* **2022**, *9*, 1–3.
4. Moniz, A.B.; Krings, B.J. Robots Working with Humans or Humans Working with Robots? Searching for Social Dimensions in New Human-Robot Interaction in Industry. *Societies* **2016**, *6*. doi:10.3390/soc6030023.
5. Zhao, W.X.; Zhou, K.; Li, J.; Tang, T.; Wang, X.; Hou, Y.; Min, Y.; Zhang, B.; Zhang, J.; Dong, Z.; Du, Y.; Yang, C.; Chen, Y.; Chen, Z.; Jiang, J.; Ren, R.; Li, Y.; Tang, X.; Liu, Z.; Liu, P.; Nie, J.Y.; Wen, J.R. A Survey of Large Language Models, 2023, [arXiv:cs.CL/2303.18223].
6. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; Agarwal, S.; Herbert-Voss, A.; Krueger, G.; Henighan, T.; Child, R.; Ramesh, A.; Ziegler, D.M.; Wu, J.; Winter, C.; Hesse, C.; Chen, M.; Sigler, E.; Litwin, M.; Gray, S.; Chess, B.; Clark, J.; Berner, C.; McCandlish, S.; Radford, A.; Sutskever, I.; Amodei, D. Language Models are Few-Shot Learners. *CoRR* **2020**, *abs/2005.14165*, [2005.14165].
7. Koubaa, A.; Boulila, W.; Ghouti, L.; Alzahem, A.; Latif, S. Exploring ChatGPT Capabilities and Limitations: A Critical Review of the NLP Game Changer. *Preprints.org* **2023**, *2023*.
8. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.u.; Polosukhin, I. Attention is All you Need. Advances in Neural Information Processing Systems; Guyon, I.; Luxburg, U.V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; Garnett, R., Eds. Curran Associates, Inc., 2017, Vol. 30.
9. OpenAI. GPT-4 Technical Report, 2023. https://cdn.openai.com/papers/gpt-4.pdf.
10. Koubaa, A. GPT-4 vs. GPT-3.5: A Concise Showdown. *Preprints.org* **2023**, *2023030422*. doi:10.20944/preprints202303.0422.v1.
11. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2018, [arXiv:cs.CL/1810.04805].
12. Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; Liu, P.J. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, 2019, [arXiv:cs.CL/1910.10683].
13. Zamfirescu-Pereira, J.D. and Wong, Richmond Y. and Hartmann, Bjoern and Yang, Qian. Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts. Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems; Association for Computing Machinery: New York, NY, USA, 2023; CHI '23. doi:10.1145/3544548.3581388.
14. Vemprala, S.; Bonatti, R.; Bucker, A.; Kapoor, A. ChatGPT for Robotics: Design Principles and Model Abilities. Technical Report MSR-TR-2023-8, Microsoft, 2023.
15. He, H. RobotGPT: From ChatGPT to Robot Intelligence **2023**.
16. Koubaa, A. ROSGPT Implementation on ROS2 (Humble).
17. Oguz, O.S.; Rampeltshammer, W.; Paillan, S.; Wollherr, D. An Ontology for Human-Human Interactions and Learning Interaction Behavior Policies. *J. Hum.-Robot Interact.* **2019**, *8*. doi:10.1145/3326539.

 doi:10.20944/preprints202304.0827.v3