Use of Assembly Inspired Instructions in the Allowance of Natural Language Processing in ROS

By

Takondwa Kakusa

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

June 19, 2018

Blacksburg, Virginia

Keywords: Natural Language Processing, Robotics, ROS

Use of Assembly Inspired Instructions in the Allowance of Natural Language Processing in ROS

by

Takondwa Kakusa

(ABSTRACT)

Natural Language processing is a growing field and widely used in both industrial and and commercial cases. Though it is difficult to create a natural language system that can robustly react to and handle every situation it is quite possible to design the system to react to specific instruction or scenario. The problem with current natural language systems used in machines, though, is that they are focused on single instructions, working to complete the instruction given then waiting for the next instruction. In this way they are not set to respond to possible conditions that are explained to them.

In the system designed and explained in this thesis, the goal is to fix this problem by introducing a method of adjusting to these conditions. The contributions made in this thesis are to design a set of instruction types that can be used in order to allow for conditional statements within natural language instructions. To create a modular system using ROS in order to allow for more robust communication and integration. Finally, the goal is to also allow for an interconnection between the written text and derived instructions that will make the sentence construction more seamless and natural for the user.

The work in this thesis will be limited in its focus to pertaining to the objective of obstacle traversal. The ideas and methodology, though, can be seen to extend into future work in the area.

Use of Assembly Inspired Instructions in the Allowance of Natural Language Processing in ROS

by

Takondwa Kakusa

(GENERAL AUDIENCE ABSTRACT)

With the growth of natural language processing and the development of artificial intelligence, it is important to take a look how to best allow these to work together. The main goal of this project is to find a way of integrating natural language so that it can be used in order to program a robot and in so doing, develop a method of translating that is not only efficient but also easy to understand. We have found we can accomplish this by creating a system that not only creates a direct correlation between the sentence and the instruction generated for the robot to understand, but also one that is able to break down complex sentences and paragraphs into multiple different instructions. This allows for a larger amount of robustness in the system.

.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Current State of the Field

Fully autonomous systems are in high demand in this day and time, with companies spending million to billions of dollars in order to meet the worlds needs through technology. The applications of this range from Tesla and Uber as they try to build the best fully autonomous vehicles, to Flippy the burger flipping robot [1], built to autonomously flip burgers. though a large focus has been placed on these sorts of technologies recently, there have been steady trends that show the needs for not just fully autonomous systems, but also semi-autonomous systems systems, requiring human interaction.

In the paper "HumanRobot Interaction Using Three-Dimensional Gestures" [2], the author describes how there is now a bridge between humans and technology, in that they no longer have to work in parallel, but rather, they can work side by side. In this, technology has developed from joysticks, to touch screens, gesture recognition and,as will be discussed in this paper, natural language processing. There are many areas in which fully autonomous systems can be necessary, such as in space with different probes, or even the autonomous robotic arm which is used in many industrial applications. At the same time there are many applications that cannot be handled by a human or a robot alone, such as medical devices and military devices that require high amounts of precision. In the medical field alone there are multiple different variations of autonomous and semi-autonomous robots being categorized as surgical computeraided design/manufacturing (CAD/CAM) systems and surgical assistants[3]. The surgical CAD/CAM systems acting more in the fully autonomous case to digitally reconstruct the body while the surgical assistants assist in the actual precision of the surgical procedure[4].

In this case, what does it truly mean for the human-robot interaction? What are the prerequisites set to ensure that the interaction is seamless? In the paper "Human-Robot Interaction in Handing-Over Tasks"[5], the authors describe seamless human-robot interaction to not require any training on the part of the human and should be intuitively simple. Never the less, there will always need to be some for of training/learning that occurs on both ends for the system to work. The main goal then becomes having a technique that allows for simplicity while minimizing the amount of training necessary on the human end.

This paper then proposes a complete instruction set that allows for easy interaction between a human and a rover through text. This natural language processing has been around for a while but with many limitations to the situations in which it can fit in as well as the full scope of the instructional capabilities. With this in mind, the main scope of this paper focuses on implementing natural language processing in order to aid in the traversal of a rover through different scenarios.

1.2 Thesis Contributions

There are two main contributions brought in from this work:

- Using N-Grams to categorize sentences in a way that extends to both basic and conditional statements.
- Design and implementation of a complete instruction set in ROS that accounts for the different possible scenarios that need to occur in the system.

1.2.1 Scope

For this research, the main goal to control a rover through plain text instructions. All the instructions that come in are expected to be commands that are in some way related to:

- The movement of the different components on the rover, such as the wheels or the camera
- Measurement of different sensors attached to the rover, such as the ultrasonic sensors
- Getting and Setting of different internal parameters of the rover, such as the robots speed

On top of this, all instructions will be assumed to be in English. With these in mind, there are many cases that are not taken into consideration when looking at sentences for processing that may be the case in other research. These will be discussed in the later chapters.

1.3 Thesis Layout

This thesis is broken down into seven different chapters ranging from a description of the current state of the problem as well as some background information, to the results of the system being designed and the conclusions gained from these results. In chapter 1, I go over some relevant information on current systems that utilize autonomous and semi-autonomous systems, pointing out why this work is necessary. Chapter 2 goes over some relevant terminology needed to understand the rest of the paper, such as what ROS is and how a set of instructions can be considered complete. Chapter 3 then goes over the different instruction types that are being used, going into detail as to where they fit in, in the system and their purpose. The next chapter, chapter 4, goes over how the text processing occurs, with the N-Gram key matching and Macros to fit the keys into their appropriate instruction sets. Chapter 5 then goes into the ROS simulation and how the network structure for the system is designed in order to generate real-time and close to real-time results both physically and virtually. The last two chapters then go over the results of running this infrastructure in multiple different situations and what conclusions can be made from these results.

Chapter 2

Literature Review

2.1 Overview

Before diving into the main topic of this thesis, a few terms and ideas need to be discussed. These set the basic understanding necessary to comprehend some of the more specific details described in later chapters. The main terms that will be looked into begin with what is meant by ”instruction completeness” and how the instruction set created for this research fits within this model. Next I go into explaining what Natural Language processing is, and how it will be used to build the instructions that will be run in ROS. Finally I will define what ROS is and the different modules and techniques inside of it that were used in the experimentation.

2.2 Instruction Completeness

An important question that comes up when looking at the fact that instructions are generated from the natural language is whether or not the set of instruction types is complete. By this we mean, can any possible case be covered within natural language, begging the question, what sort of sentences are expected. Because of the scope of the thesis, instructions cannot be open ended and are expected to fit into the context of the goals of the rover, which in this case, is getting from point A to point B, in a manner that is safe. Therefore the different criteria that need to be met, and for these criteria we look at ”The Instruction Set Completeness Theorem” [6] which states the following cases:

1. **Statefulness:** The instruction set must be stateful by state and a next state function
2. **Explicit State Change:** There must be an instruction to directly and explicitly modify state.
3. **Explicit Next State Change:** There must be an instruction to directly and explicitly modify next state function.
4. **Implicit State Change:** There must be an instruction to indirectly and implicitly modify state.
5. **Implicit Next State Change:** There must be an instruction to indirectly and implicitly modify next state function.

In these cases, explicit means that the operation is intentionally made by the user, while implicit means that the operation is automatically generated. Since four of the principles center around implicit and explicit state changes, the theorem is then simplified into the following three theorems [6]:

1. **Statefulness:** The instruction set must be stateful by state and a next state function
2. **State Change Principle:** There must be an instruction to modify state.
3. **Next State Change Principle:** There must be an instruction to modify next state function.

Though these come from the stand point of looking at things in the eye of computer architecture, they seem fitting to the current ideas that are being pursued. This is mainly due to the fact that both the natural language processing and, especially, the ROS instruction base is very much state based. Using the ideas listed here, we can form a basis for the completeness of the instruction sets that are available for use in this thesis.

2.3 Natural Language Processing / Programming

Natural Language Processing (NLP) is the step towards machine learning and artificial intelligence as it tries to get the computer to understand human speech, therefore becoming more human. In essence, the goal of NLP is to get computers and humans to communicate much more easily, with less learning needing to occur on the human end to accommodate the technology. As an extension of this, Natural Language Programming (also NLP) is a subset of this type of research that focuses more on trying to get computers to understand programming instructions in terms of natural text rather than through a specific programming language. There are many tools that try to do this to varying degrees within the scope of their applications[7][8], but many of them do not follow the cause and effect approach found in this research, making them more direct instruction oriented. Due to the large variability in sentences, there are many aspects that need to be taken into consideration for either of these to become successful that people take for granted. Here are a few key components that computers must understand to get the full depth of knowledge of sentence structure[9]:

- **Phonological:** How words sound when when we hear them (more on the end of speech recognition)
- **Morphological:** The ability to break down a word into its basic units, or morphemes, in order to get the root understanding. For example, taking the word "worldly" and understanding its built off of the root "world" and the extension "ly".
- **Syntactical:** The word organization and how they come to form a complete sentence.
- **Pragmatics:** Understanding the context in which the sentence is used and using the context to alter decisions on the overall meaning.
- **World:** Understanding of the other environmental factors such as the other persons beliefs and goals.

The paper "Intelligent Natural Language Processing"[10] simplifies this even more by breaking these factors into three separate categories:

- **Knowledge about words:** Basic understanding about what words mean.
- **Knowledge about sentences:** Knowledge of how the words are put together.
- **Knowledge about discourse:** Understanding over how the sentences are used, such as to ask questions, make commands and convey emotions.

With all of this in mind, it is essential to point out that having a universal NLP algorithm is far from being established and is not the goal with this research. Like other research and work that goes into this field, a certain scope needs to be set on what sort of information is expected and how much the researchers plan to process that information. Using the scope defined in the previous chapter, there are several simplifications that can be made to these components in order to make the processing easier.

2.3.1 Language Structures

A lot of research has been put into looking at the feasibility of using natural language for programming[11][12][13]. All of this research starts by looking into the structure of the language and using details on that front to help derive the rules that need to be generated. Most of these follow the general principles of looking at the following[11]:

- **Syntactic Typing:** This component is the basis of most languages and helps solve many ambiguities between sentences. If focuses on looking at the parts-of-speech (noun, verb, abjective, etc.) and using that as the basis of sorting the wording and defining meanings.
- **Inheritance:** Characteristics that can be determined by other factors give, like having a sub-class from a base-class.
- **Reference:** Using other objects in a sentence to relate to a specific object. In this research, all references will be made with respect to the rover, such as "turn the rover left" refers to the rover's left rather than that of the person giving the instruction.
- **Conditionals, Iteration, and Stepping:** Researchers have found that forming these types of statements does not occur the same way naturally as it may happen programmatically[14]. Most sentences are naturally flattened out into what are called

double-clause structures, basically a cause and effect form. This is essential to look at as the research will focus on this relationship to drive its instructions.

Looking at these, there are many aspects that need to be focused on for the sentence structure and combining many of these aspects can lead to a deep understanding of conversation being had.

2.3.2 Processing Techniques

There are many techniques that can be used in order to process the text, each with their own benefits and faults. The four major ones are[10]:

- **What you see is what you get (WYSIWYG):** A top down look at the system. Breaks down into four modules, one for the each component discussed before (word and sentence knowledge, semantics and pragmatics).
- **Oatmeal:** Rather than separating out all of the components, it draws a box around them all as it tries to encompass all the aspects at once. This is a very non-modular approach.
- **Heterarchy:** A modular approach that creates a centralized node that directs the information to the following modules.
- **Pipelined:** Another modular view in which the states are separated out like they are in the first case, but the information can pass back and forth between them to gain more information from other sources.

For this research, the WYSIWYG approach was utilized in order to have a more logical view of how the information was being passed from one state to the other. Particularly with this, an emphasis was placed on *Set-Based Dynamic Reference*[15]. In this methodology, I used sets and set based operations in order to interpret the semantics given within the sentence. The combination of these two factors keeps the processing logical and and visually simple to understand and adjust when necessary.

There are then three main approaches that come into play when looking at the sentiment analysis itself in each of the stages[16]:

- **Machine Learning Approach:** Both the use of supervised and unsupervised learning techniques such as SVM's and Native Bayesian classification.
- **Lexicon Based Approach:** This is divided into two realms. The first is a dictionary based approach in which a universal dictionary is used, such as one generated from WordNet or SentiWordNet. This takes in the opinion words from the text and uses the dictionaries to narrow down the meaning from common sysnonyms and antonyms. the second method is a corpus based method that is more context specific as the word are related to a predefined corpus specific to the project/research.
- **Hybrid Approach:** Uses the lexicon method for sentiment scoring and then uses the machine learning on the new trained data to perform the analysis.

Following this same trend of logical visibility, the approach of a corpus based N-Gram method was chosen. In this, a dictionary of key terms was created as they related to the scope of the project and these terms were divided according to their syntactic attachments.

Finally, with this system in mind, we have an idea of the approach we can take for the analysis and can divide that analysis into the components of[15]:

1. **Step finder:** Identifying the actions that need to be taken.
2. **Loop finder:** Identify which instructions indicate a repetition that may occur.

2.3.3 Python and NLP

Python[17] is a programming language with a wide range of capabilities in higher level programming. Its uses vary wildly from high level data processing[18], machine learning[19], and even robotics [20]. Its largest benefits are the fact that its easy to get started, friendly and easy to learn, open-source, and, most importantly, host thousands of third-party modules[17]. With all of this, the largest cost comes in terms of performance, and case studies have been made to look at how this could effect the overall processing for both ML and NLP[21]. In the end they do find that the language supports a wide range of methods for both of these applications.

Python was chosen because of these factors and its inclusion of the NLTK libraries[22] which are a set of libraries dedicated to machine learning and natural language processing. This toolkit was developed in conjunction with a computational linguistics course at the University of Pennsylvania and contains a large set of modules and corpora[23].

2.4 Robot Operating System (ROS)

ROS stands for Robot Operating System and is a collection of tools, libraries, and conventions used to simplify the task of creating complex and robust behavior [24]. Its popularity stems not only from the fact that it is open source, but that it is also robust enough to be used in both simple and commercial products. The ROS website [24] talks about the difficulties of creating a robust enough software system for a robot, and part of this is due to the fact that situations that may seem easy for humans could vary wildly in terms of the instance of the task and the environment. These sorts of variations are hard for any single person to develop around, and thus the Robot Operating System was developed to help mitigate this, through its focus on the connections between embedded systems within the hardware stack. Also through its focus on real time motion to push designers to partition robots into real-time and non-real-time subsystems[25].

ROS began as a collaboration between Willow Garage and the Stanford robotics department in 2007[26] and has since grown to be the leading universal operating system used in large scale robotics projects. The growth of ROS has become quite exponential since its roots and this has been measured in two different ways. The first is that the number of public repositories that contain ROS code has grown exponentially with more and more institutions contributing open-source packages to ROS[27]. The other parameter that shows this growth is the number of unique ROS packages that have been created, and this has seen a spike with the creation of new point cloud sensors, like those used in the Kinect, by Microsoft, by allowing for more varied 2D and 3D image data[27]. This growth is greatly due to the wide integration and support ROS has gained from not only academia, but from major companies in the robotics field such as SRI and Gostai, with government sponsors like DARPA advocating for its continued use [27]. With all this in mind, this is why this system was chosen as a means of pioneering this research. Now lets dive into how ROS works.

2.4.1 ROS Design Principles

In the paper "ROS: an open-source Robot Operating System", the authors go into detail about the five philosophical goals that ROS tries to achieve. These are being peer-to-peer, tools-based, multi-lingual, thin, free and open-sourced[28]. Each of these are described briefly below:

1. **Peer-to-Peer:** ROS is built to contain multiple processes, most likely on many separate hosts, that are connected at runtime. These hosts are connected in some sort of

LAN configuration in order to avoid slow wireless links and uses a dedicated lookup mechanism to allow the processes to find each other at runtime.

2. **Multi-lingual:** ROS can be used with multiple different programming languages and is in general language neutral. Currently ROS supports C++, Python, Octave, and LISP with others still in various stages of development. This is done by defining an interface definition language (IDL) that sets a neutral standard in which these different languages can communicate.

3. **Tools-Based:** In attempt to battle the large complexity that can be obtained when designing large systems, the designers opted for a microkernal design. In this, they developed a large number of small tools and applications to run their various systems rather than one large overbearing tool to control them all. This, though it may lose some efficiency, is more future proof and robust..

4. **Thin:** In an attempt to allow code and algorithms used in ROS to be implemented in other non-ROS projects, the developers focused on pushing users to create libraries that are ROS independent. These libraries are then tied into the ROS system through a simple CMake make command, allowing for a more universal use of these algorithms.

5. **Free and Open-Source:** One of the largest benefits of ROS is that it is open source, meaning that there are thousands of people contributing to its development at any time. Though there are other non-open-source comparisons such as Microsoft Robotics Studio[29] and Webots[30], they are limited when it comes to catching up to new developments in technology and are scoped to the single design given to them.

2.4.2 ROS Terminology

There are a few terms that are used in this paper to describe actions and structures inside of ROS. These terms will be defined here in order to have a clear basis to work off of:

- **Nodes:** ROS works by facilitating the communication between multiple different nodes. These nodes perform the computations that are necessary for the robot, such as sensor reading and processing as well as data and information calculations. Thank to the multi-lingual nature discussed before, these nodes can be written in multiple different languages as long as they follow the appropriate message passing parameters[28].
- **Messages:** The way in which nodes communicate with each other. They are designed to use primitive types, such as integers and booleans, and are strictly typed data structures[28].
- **Topic:** A location in which nodes can publish data that will be read in by other nodes. multiple nodes can subscribe and publish to a single topic without knowing the existence of the other nodes[28].
- **Publisher:** A node is considered a publisher if it sends a message to a topic.
- **Subscriber:** A node is considered a subscriber if it reads messages from a topic.
- **Service:** A service within a node is a routine that gets accessed by another particular node. This is normally in a blocking manner.
- **Client:** A node that calls on a service from another node is considered a client.

2.4.3 ROS Control

As stated in the ROS Design Principles section, ROS is tools based, meaning that it relies on multiple smaller packages to make the system work. This idea extends to how ROS handles its control algorithm, through the ROS controls package, which can be seen in figure 2.1[31].

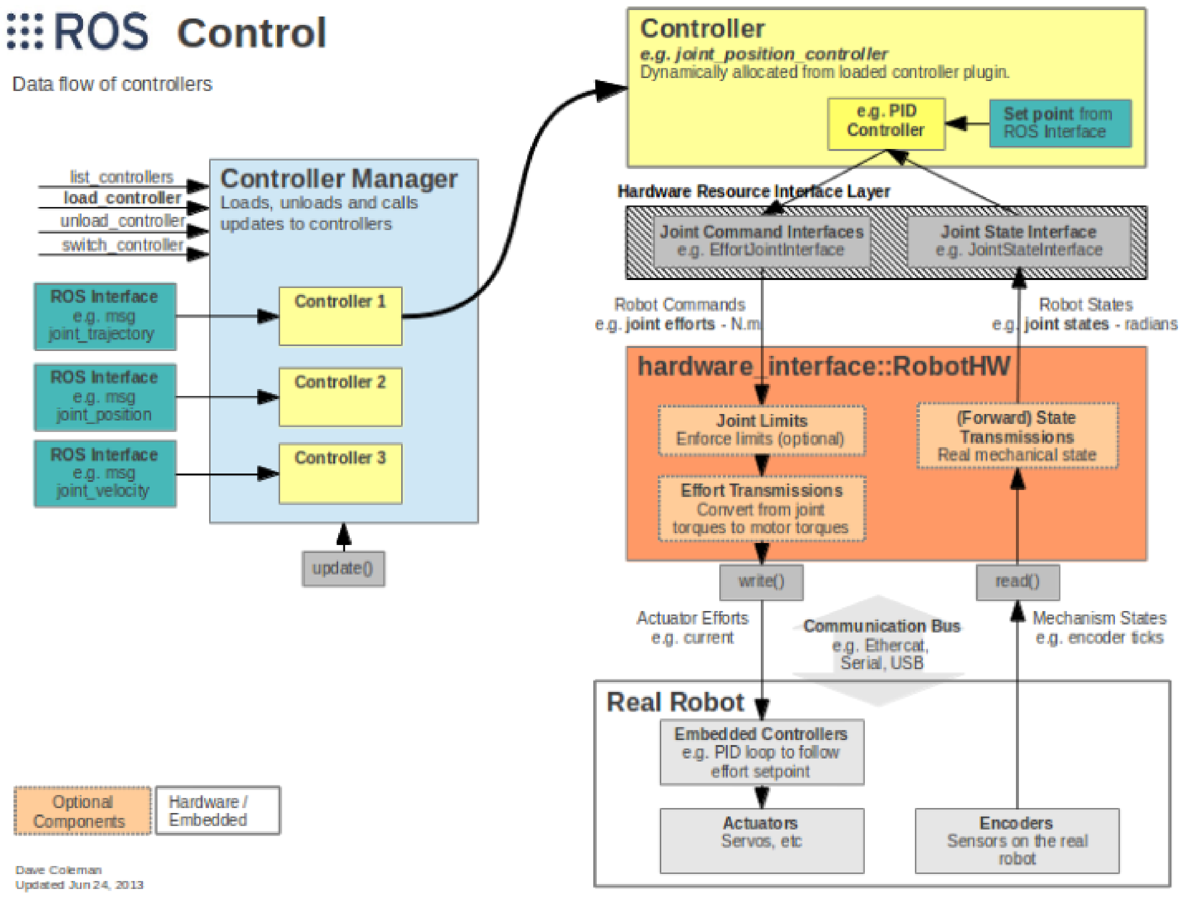

ROS Control
Data flow of controllers
list_controllers
load_controller
unload_controller
switch_controller
Controller Manager
Loads, unloads and calls updates to controllers
ROS Interface e.g. msg joint_trajectory
ROS Interface e.g. msg joint_position
ROS Interface e.g. msg joint_velocity
Controller 1
Controller 2
Controller 3
update()
Controller
e.g. joint_position_controller
Dynamically allocated from loaded controller plugin.
e.g. PID Controller
Set point from ROS Interface
Hardware Resource Interface Layer
Joint Command Interfaces e.g. EffortJointInterface
Joint State Interface e.g. JointStateInterface
Robot Commands e.g. joint efforts - N.m
Robot States e.g. joint states - radians
hardware_interface::RobotHW
Joint Limits Enforce limits (optional)
(Forward) State Transmissions Real mechanical state
Effort Transmissions Convert from joint torques to motor torques
write()
read()
Actuator Efforts e.g. current
Communication Bus e.g. Ethercat, Serial, USB
Mechanism States e.g. encoder ticks
Real Robot
Embedded Controllers e.g. PID loop to follow effort setpoint
Actuators Servos, etc
Encoders Sensors on the real robot
Optional Components
Hardware / Embedded
Dave Coleman
Updated Jun 24, 2013

Figure 2.1: ROS Control: Data Flow of controllers

The ros_control packages takes the joint state information from the robots actuators and encoders as inputs. Using a PID controller, it takes these inputs and controls the output, usually in term of effort[31]. with this sort of package structure, there are four main goals that are trying to be achieved. First, it tries to provide a lower entry barrier for exposing hardware to ROS. Nest it tries to promote reuse of control code and provide ready-to-use tools. Finally it serves to provide Real-time ready implementation[32].

2.5 ROS and Gazebo

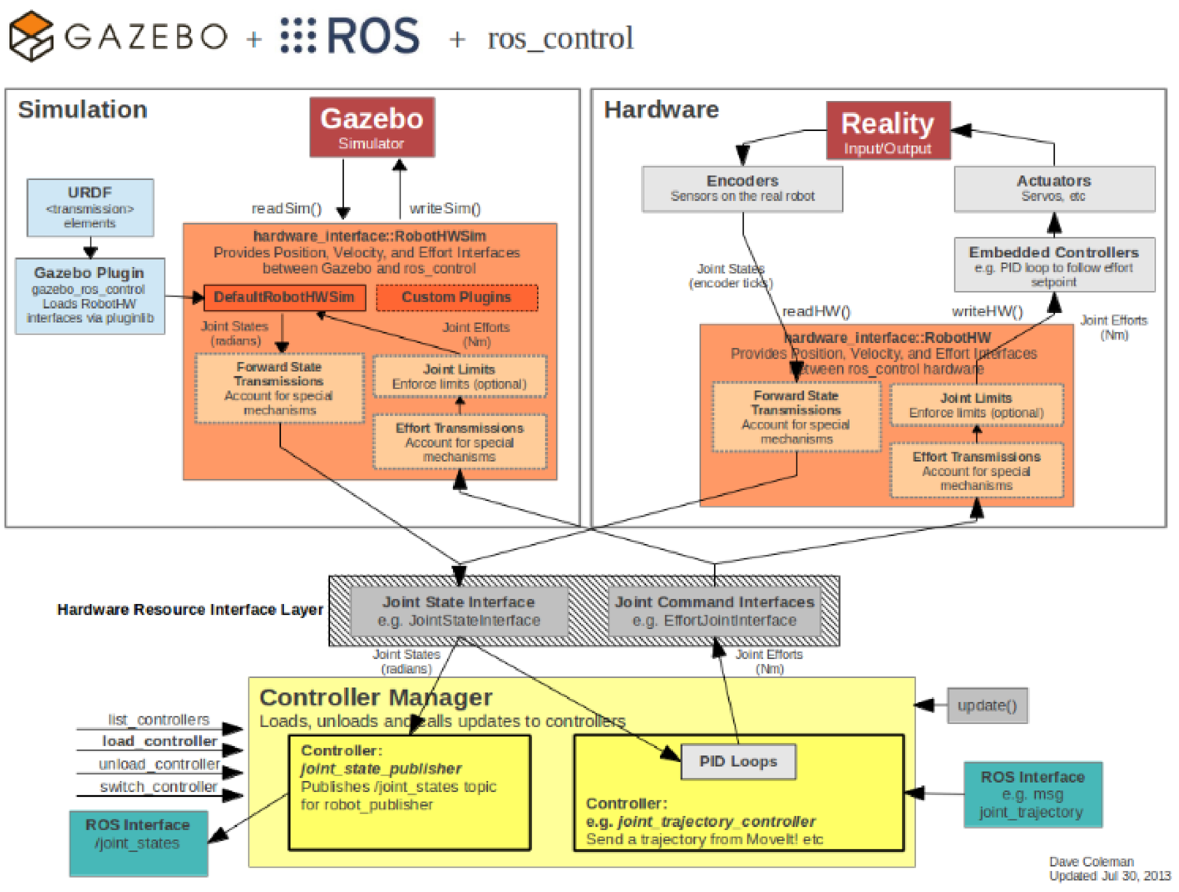

GAZEBO + ROS + ros_control
Simulation
Gazebo
Simulator
URDF
<transmission> elements
Gazebo Plugin
gazebo_ros_control
Loads RobotHW interfaces via pluginlib
readSim()
writeSim()
hardware_interface::RobotHWSim
Provides Position, Velocity, and Effort Interfaces between Gazebo and ros_control
DefaultRobotHWSim
Custom Plugins
Joint States (radians)
Joint Efforts (Nm)
Forward State Transmissions
Account for special mechanisms
Joint Limits
Enforce limits (optional)
Effort Transmissions
Account for special mechanisms
Hardware
Reality
Input/Output
Encoders
Sensors on the real robot
Actuators
Servos, etc
Embedded Controllers
e.g. PID loop to follow effort setpoint
Joint States (encoder ticks)
readHW()
writeHW()
Joint Efforts (Nm)
hardware_interface::RobotHW
Provides Position, Velocity, and Effort Interfaces between ros_control hardware
Hardware Resource Interface Layer
Joint State Interface
e.g. JointStateInterface
Joint Command Interfaces
e.g. EffortJointInterface
Joint States (radians)
Joint Efforts (Nm)
Controller Manager
Loads, unloads and calls updates to controllers
list_controllers
load_controller
unload_controller
switch_controller
update()
Controller:
joint_state_publisher
Publishes /joint_states topic for robot_publisher
PID Loops
Controller:
e.g. joint_trajectory_controller
Send a trajectory from MoveIt! etc
ROS Interface
/joint_states
ROS Interface
e.g. msg joint_trajectory
Dave Coleman
Updated Jul 30, 2013

Figure 2.2: Integration of Gazebo and the ros_control packages

Gazebo is an open-source simulation environment, and is used in this research to simulate the movements of the rover. The reason Gazebo is chosen is due to how closely tied it is to the mechanisms inside of ROS, to the point that transitioning between software simulation and actual hardware testing is made a lot easier. Figure 2.2[33] shows how the integration of Gazebo and the ros_control packages work together. In this you can see the Hardware Resource Interface Layer, where all the joint states and commands are housed. In this layer is where Gazebo is able to send its simulated information of the hardware to ROS, thereby emulating the control signals sent by the actual hardware.

2.6 OpenCV

Since computer vision is not a main focus of this research and is, therefore, not touched too much throughout the work I will only go into a brief description of the OpenCV libraries and why they were used. As stated in the paper ”Realtime Computer Vision with OpenCV”[34], there is a growing need for image processing in an attempt to determine what is going on in front of the camera, so that we can use that information to control robots. The main problem being that doing any sort of work with images is very computationally expensive and sometimes requires a compromise in quality in order to make the entire system run[34]. OpenCV is then a group of libraries built to help with image processing, and is built mainly in C++ and python. The reason we chose to use OpenCV and do image processing in general was to test more real-life scenarios in which actions done by the robot are based off of what it can see and detect in the real world.

Chapter 3

Instruction Types

3.1 Overview

Having a robust set of instruction types is key to the full development of this system. The challenge then is creating an instruction set that not only handles the current cases but is universal and robust enough to be future proofed. On top of this, the instruction sets need to make up just a small enough piece of the actions that need to be taken so that they can easily be concatenated for the most efficient instruction traversal.

The base design of this procedure is built upon the use of different instruction types as the building block for all the possible instructions. There are five main instruction types, GET, SET, RUN, CHECK and LOOP instructions, with some having different variations depending on what is necessary. Each of these instructions are generated based on the particular wording used as the input (this is described more in more detail in chapter 4).

Each of the sections in this chapter will go over the different instruction type. For each, a description of the use case will be laid out, a model of the structures design will be shown with the components explained, and an example will be displayed.

3.2 Basic vs. Conditional Instructions

Before going into the individual instructions, it would be good to note here that the instructions are broken into two different types, basic and conditional.

1. **Basic Instructions:**
   - Simple instructions that do not require more than a single step or instruction to be fully complete.
   - The GET, SET, and RUN instructions all fall under this category as they all can be accomplished within a single instruction.
   - In the context of natural language, these instructions relate more to actions that need to be taken. For example "The car should move forward" is a RUN instruction that gives a forward movement action causing the car to move. Though GET and SET instructions are more internal than external, in the same vein the instruction "Set the speed of the car to 50" causes actions internally to change the car speed.
2. **Conditional Instructions:**
   - Instructions that can take up multiple cycles depending on the instructions that surround them as well as the conditions that are set on them.
   - These instructions are what allow for the allowance of conditional statements in the program.

- These instructions can grow as well and can be placed within each other for even further expansion.
- They must use a terminating key in the text processing stage in order to signify the end of the instruction.
- CHECK and LOOP instructions are both types of conditional instructions.
- In the natural language context, these are meant to take the case of checks in the system, that lead to the next actions that should be taken. For example the instruction "if the car is less than 30 inches from the wall, stop" causes the system to analyze the current state of the car and route the appropriate response due to this. No direct actions are taken from conditional instructions, like they are for basic instructions, but they are essential for the overall system routing of conditions in the natural language.

3.3 Base Design

3.3.1 Description

Each of the instructions share some similar qualities that are used to index them as well as allow for transitions between instructions. This base design is essential to the entire system and expected in order to meet the minimum requirements of the instructions

3.3.2 Structure

Parameters for subsection 3.3.2

- **tier:** used in the case of multiple concatenated conditions in order to tell statements

```
base_instruction = {
        "tier": tier,
    "inst_type": inst_type,
    "inst_count": inst_count,
    "inst_next": inst_next,
    "inst_fail_next": inst_fail_next,
    "parameters": {
        "count":0,
    }
}
```

Figure 3.1: Base Design Structure

that are on the same level

- **inst_type:** Used to define the type of instruction that is going to be used (GET, SET, RUN etc.).
- **inst_count:** Keeps track of the location of this instruction and is unique to each instruction. Value also helps in determining the order of the instructions.
- **inst_next:** Used to tell what the next instruction will be after the completion of the current instruction. In the case of the CHECK and LOOP instructions this is used if the statement being evaluated comes out to be true.
- **inst_fail_next:** Used only in CHECK and LOOP instructions. This is used in the case that the statement being evaluated comes out to be false. For the basic instructions, the inst_next value is copied over into this section.
- **parameters:** This section is used differently depending on the instruction type, and will be further detailed in the other sections.
- **count:** Used mainly by the Basic instructions, the count identifies the number of parameters that need to be complete before the instruction is considered accomplished.

3.4 GET Instructions

3.4.1 Description

GET instructions go under the category of basic instructions and are used to take received sensor data and store it in a specified variable. This can be proven necessary in the cases where the condition of the sensor could differ at different states of the instruction process. This also becomes useful when a single sensor needs to be used for multiple different tasks. It is important to note that this instruction is run at the moment that it is activated, meaning that no other instructions are being run while this instruction is being executed.

3.4.2 Structure

```
base_instruction = {
        "tier": tier,
    "inst_type": get,
    "inst_count": inst_count,
    "inst_next": inst_next,
    "inst_fail_next": inst_fail_next,
    "parameters": {
        "count":0,
        "node": node,
        "get_param": get_param,
    }
}
```

Figure 3.2: GET Instruction Structure

Parameters for subsection 3.4.2

- **node:** Used to identify the node that needs to be communicated with in order to get the sensor value necessary.

- **get_param:** Indicates the parameter that needs to be obtained, therefore indicating the variable that this will be stored in.

3.4.3 Example

A good example of this can be seen in the use of the ultrasonic sensor of the rover. Since the sensor is not omni-directional, such as a lidar, it can only report the distance from one direction. Thus, rather than using three separate sensors for each direction, it utilizes the pan and tilt assembly it is attached to in order to rotate the sensor left, right and forward and the get instruction is used to get the specific value as follows:

```
base_instruction = {
        "tier": tier,
    "inst_type": get,
    "inst_count": 0,
    "inst_next": 1,
    "inst_fail_next": 1,
    "parameters": {
        "count":0,
        "node": ultrasonic,
        "get_param": left_distance,
    }
}
```

Figure 3.3: GET Structure Example

In the above instruction, the left distance is obtained and stored inside its own variable after polling the node for the ultrasonic sensor.

3.5 SET Instructions

3.5.1 Description

SET instructions are used in a similar way as GET instructions but rather than waiting to receive and store a value, a value is sent to a specific node. This is mainly used in order to modify/adjust a specific parameter within a node in order optimize the functionality. Also like the GET instruction it is run at the moment that it is activated, meaning that no other instructions are being run while this instruction is being executed.

3.5.2 Structure

```
base_instruction = {
        "tier": tier,
    "inst_type": set,
    "inst_count": inst_count,
    "inst_next": inst_next,
    "inst_fail_next": inst_fail_next,
    "parameters": {
        "count":0,
        "node": node,
        "set_param": set_param,
        "value": value
    }
}
```

Figure 3.4: SET Instruction Structure

Parameters for subsection 3.5.2

- **node:** Used to identify the node that needs to be communicated with in order to set the value parameter.

- **set_param:** Indicates the parameter that needs to be set within the given node.
- **value:** The value that the parameter will be set to within the node. The value can be either a string or number depending on the module being looked into.

3.5.3 Example

A definite use case for this instruction type is when determining the speed at which the rover should move at. The user may want to adjust this according to the environment or other variable that may be present as well as just due to just personal preferences. An example of this can be seen below:

```
base_instruction = {
        "tier": tier,
    "inst_type": set,
    "inst_count": 0,
    "inst_next": 1,
    "inst_fail_next": 1,
    "parameters": {
        "count":0,
        "node": movement,
        "set_param": speed,
        "value": 50
    }
}
```

Figure 3.5: SET Structure Example

In the above example the speed of the rovers movement is changed to 50 (movement speed ranges from 0 to 100).

3.6 RUN Instructions

3.6.1 Description

RUN instructions differ from the previous GET and SET instructions as in most cases it both sends and wait to receive information to and from the other sensor nodes. The information that is sent is used to start or stop a particular action and the information that is waiting to be received is used towards signifying the end of the instruction. Due to the variability of this type of instruction, the length of its parameters is not consistent.

3.6.2 Structure

```
base_instruction = {
        "tier": tier,
    "inst_type": run,
    "inst_count": inst_count,
    "inst_next": inst_next,
    "inst_fail_next": inst_fail_next,
    "node_data_1": {
        "extra_parameter_1": extra_parameter_1,
        ...
    }
    "parameters": {
        "count": count,
        "node_1": {
            "status": status,
            "achieved": achieved,
        },
        "node_2": {
            ...
        }
    }
}
```

Figure 3.6: RUN Instruction Structure

Parameters for subsection 3.6.2

- **node_data:** Used to indicate the node to which the data needs to be sent to.
- **extra_parameter:** Data that needs to be sent to a particular node to start the specific action that needs to be accomplished. The information and the amount of information sent can differ depending on what needs to occur and the node being targeted.
- **count:** In this case the count is used to indicate the number of parameters that need to be complete for the instruction to be considered complete. The count is equal to the number of active parameters in the dictionary.
- **node:** The nodes (node_1 and node_2) indicate the nodes to which the parameter data needs to be sent to and tracked from.
- **status:** States whether or not the node needs to be tracked for a result or not (active or inactive). If active, the achieved value is looked into.
- **achieved:** Used when the parameter is active and starts off at false. Once a message is received from the particular node that the information was sent to and the information matches the parameter expected the state is changed to true.

3.6.3 Example

A bulk of the instructions that are generated use this form of instruction and this is mainly due to its high variability. In this example a simple motor turning instruction is used to demonstrate one of the possible options:

In the example above, a simple "turn the car left" instruction is used. In this case the turn duration is dictated by a single timer. It can be seen from this, the "movement" node

```
base_instruction = {
        "tier": tier,
    "inst_type": run,
    "inst_count": 0,
    "inst_next": 1,
    "inst_fail_next": 1,
    "movement": {
        "direction": 'left',
    }
    "timer": {
        "inst_count": 0,
        "time": 2.0
    }
    "parameters": {
        "count": 1,
        "time": {
            "status": 'active',
            "achieved": False,
        },
    }
}
```

Figure 3.7: RUN Structure Example

data has just the single extra parameter of "direction" while the timer node data has two parameters. This shows how different nodes can have varying lengths of data that they need to send. In this case the timer needs to send the instruction count along with the time it needs to count since the instruction count is used when the timer is finished to identify the instruction that sent the original request. This is explained in more detain in chapter 5.

3.7 CHECK Instructions

3.7.1 Description

This instruction acts as an IF statement, and is used to compare two values to each other. This is the first type of conditional instruction and is essential in allowing for branching depending on the conditions given. Overall, the complexity does not come in when creating the instruction, but rather in how the instruction is integrated with the other instructions around it. The integration is further explained in chapter 4 but it is important to note that there three types of statements that fall into this category.

The first is the check instruction itself which acts as an IF statement depending on how it is placed. The second is called a check_alt instruction and this is mainly to act as the alternative to the previous check statement, in the same way as an ELSE IF statement. The final type in the check_fin instruction and is mainly used as a place holder to work as an ELSE instruction. The structure section below only shows the structure for the check instruction since the check_fin instruction in completely identical to the base structure and the check_alt instruction is identical to the check instruction.

3.7.2 Structure

Parameters for subsection 3.7.2

- **inst_fail_next:** Used to jump to the next appropriate instruction in the case that the condition that being checked returns as false.
- **param_1:** First parameter used in the comparison, used on the left side of the comparison.

```
base_instruction = {
        "tier": tier,
    "inst_type": check,
    "inst_count": inst_count,
    "inst_next": inst_next,
    "inst_fail_next": inst_fail_next,
    "parameters": {
        "count": 0,
        "param_1": param_1,
        "param_2": param_2,
        "equality": equality
    }
}
```

Figure 3.8: CHECK Instruction Structure

- **param_2:** Second parameter used in the comparison, used on the right side of the comparison.
- **equality:** Used to determine how the two parameters are to be compared to each other (less than, greater than or equal to).

3.7.3 Example

As stated before, this instruction is meant to be used in the case that it is necessary to decide on the next instruction that needs to be done, based on a specific condition. The example in subsection 3.7.3 shows a simple check of if the distance to the wall/obstacle is less than 20 inches away. It should be noted that the inst_fail_next set in the example assumes that there is only one statement within the IF statement, this can change depending on the input given.

```
base_instruction = {
        "tier": tier ,
    "inst_type": check ,
    "inst_count": 0,
    "inst_next": 1,
    "inst_fail_next": 2,
    "parameters": {
        "count": 0,
        "param_1": "distance",
        "param_2": 20,
        "equality": less ,
    }
}
```

Figure 3.9: CHECK Structure Example

3.8 LOOP Instructions

3.8.1 Description

There are two different types of looping instructions that are used in the program in order to serve separate types of functionality. The first of these is the normal ”loop” instruction type and this serves as a WHILE or FOR loop type. Like the CHECK instructions before, this instruction type checks the conditions listed in its parameters and uses the result of these to decide the next instruction to be sent out. The main different is that, while the condition is true, the instructions will keep looping until the condition returns false. This allows for multiple instructions to run continually in a loop until the condition fails.

The next type of looping instruction is the ”loop_wait” instruction type, and this instruction has a similar blocking idea as that of the RUN instructions. The structure of the loop_wait instruction is equivalent to that of the base loop instruction, with the only difference being in how they block. Unlike the loop instruction, this instruction does not allow for other statements to be run before it is complete. Therefore, while the condition in the statement has not failed, the robot will be relying on the previous commands that have run until the condition is failed.

3.8.2 Structure

```
base_instruction = {
        "tier": tier,
    "inst_type": loop,
    "inst_count": inst_count,
    "inst_next": inst_next,
    "inst_fail_next": inst_fail_next,
    "parameters": {
        "count": 0,
        "param_1": param_1,
        "param_2": param_2,
        "equality": equality
    }
}
```

Figure 3.10: LOOP Instruction Structure

Parameters for subsection 3.8.2

- All parameters the same as the CHECK structure parameters

3.8.3 Example

The example below subsection 3.8.3 displays the use of the loop_wait instruction type. In this instruction, the text that it has been translated on states "when the car is 30 inches from the wall". This is a clear example of where the loop and loop_wait instruction types differ. If this were a loop instruction, the condition would be checked and if it passed, the next instruction in the loop would be implemented. In the case of the example, though, the instructions will not move on until the condition is failed. A statement like this, in essence, should be preceded by an action that will allow for the statements completion. In the case of the example below, an instruction like "move forward" should precede this instruction.

```
base_instruction = {
        "tier": tier,
    "inst_type": loop_wait,
    "inst_count": 0,
    "inst_next": 1,
    "inst_fail_next": 1,
    "parameters": {
        "count": 0,
        "param_1": "distance",
        "param_2": 30,
        "equality": less,
    }
}
```

Figure 3.11: LOOP Structure Example

3.9 Completeness

As stated in section 2.2, the criteria being used to check for the completeness of the instruction set comes from "The Instruction Set Completeness Theorem". Using this we can identify how each of the instruction types fit in.

1. **Statefulness Principle:** The instruction set must be stateful by state and a next state function.

   The idea of states can be seen in the instruction ordering itself. As the instructions are ordered sequentially, each instruction acts as the current state of the system. For example, if the instruction "The car should move forward for 2 seconds, then stop" were given, then two instructions would be generated. One for the first half to start the movement and the timer, and another for stopping the vehicle. In this case the first instruction is the initial state, and, once completed, will move on to the next state.

2. **State Change Principle:** There must be an instruction to modify state.

   Explicit and implicit examples of this can be found with the SET and GET commands, as users can use these to change or acquire the specific state of any of the other available nodes inside the system. The RUN Command also does this same action as it changes the states of the nodes depending on the action(s) that need(s) to occur. For example the statement "turn left for 3 seconds" will change the state of the movement node from "stop" to "left" as well as turn the timer from "off" to "on".

3. **Next State Change Principle:** There must be an instruction to modify next state function.

   This idea can be seen in the way state transitions are handled in all the instruction types through the "inst_next" and the "inst_fail_next" tags that are held within. These modify the state of the instructions by identifying what the next state will be given that type of instruction. These would be viewed as an implicit means of creating transitioning between the instructions as the user has no direct control, in the case of the basic instructions. The conditional instructions, on the other hand, can be viewed as the explicit way of transitioning between states. This is due to the fact that users can use the CHECK and LOOP instructions to modify the natural liner flow of the instructions to fit the conditions that they need.

As it can be seen from the aspects listed above, the instruction set can be seen as complete. Basically, this is due to its stateful nature, as well as its ability to implicitly and explicitly handle the current state and the next state functions.

3.10 Summary

The use of these base instructions build upon the robustness of the system as they allow for different permutations in the possible actions that can be done by the robot. Through the variety of instruction types, instruction completeness can be achieved allowing for a robust set of permutations. The instruction set described in this section is enough to handle the situations needed by the rover as well as the variety of modulations that natural language could bring in.

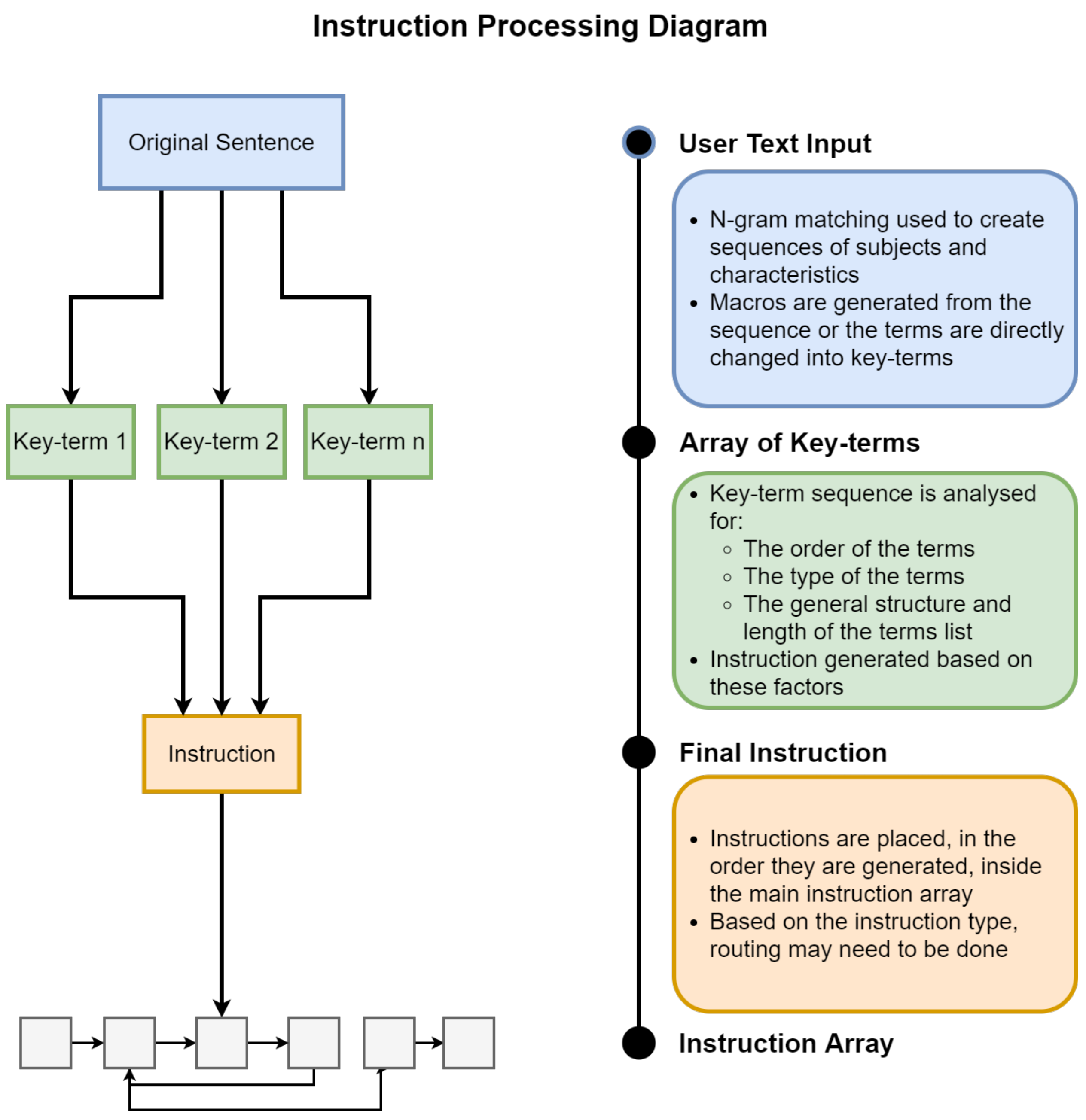
Chapter 4

Text Processing Layout

4.1 Overview

During the text processing stage, the main challenge now becomes finding a way of taking these natural sentences and not only matching them to plain instructions, but also allowing for routing conditions for the conditional statements. Not only must syntax and grammar be looked into, but the overall positioning and pairing of key terms, and how this effects the type of instruction that is generated.

During the text/natural language processing, there are a few stages that are needed in order to ensure that the proper instructions are obtained. A large part of this comes from needing to allow for conditional statements, meaning that certain types of parameters must be placed around defining the beginning and end of these conditions. The basic steps start with obtaining the keywords from the sentences, using NGRAM matching to generate macros, if necessary, and generating the instruction from the given NGRAM options.If an instruction cannot be created an error is thrown. The full visual of this can be seen in figure 4.1.



Instruction Processing Diagram
Original Sentence
Key-term 1
Key-term 2
Key-term n
Instruction
User Text Input
• N-gram matching used to create sequences of subjects and characteristics
• Macros are generated from the sequence or the terms are directly changed into key-terms
Array of Key-terms
• Key-term sequence is analysed for:
∘ The order of the terms
∘ The type of the terms
∘ The general structure and length of the terms list
• Instruction generated based on these factors
Final Instruction
• Instructions are placed, in the order they are generated, inside the main instruction array
• Based on the instruction type, routing may need to be done
Instruction Array

Figure 4.1: Instruction Processing Diagram

4.2 NGRAM Key Processing

During this stage of the sentence processing, the main goal is to take the sentence and form it into a keyword string called an NGRAM and then parse the instruction type from it. Since the goal is to make the text as natural as possible, the program must be able to manage paragraph structures with conditionals embedded inside them in some way. In order to do this, not only does the sentence structure matter, but so does the order in which sentences appear and the terms used to conjoin specific parts.

4.2.1 Converting Paragraphs to key terms

Before key-terms can be obtained, the text document with all the instruction sentences must be taken in and broken down into paragraphs, and then again into sentences. In order to do this, the python package called nltk was used, as it had the pre-requisite code for distinguishing the true endings of sentences vs false positives that could be given within the sentence.

Once the sentences have been obtained, checks are made to see if any of the following statements apply:

- Conditional Statement: This check is to see if the statement contains keywords that would have the program assume it will be a conditional statement that can take in multiple instructions. This is important, as it is necessary to establish the beginning and the end of these types of statements in order to obtain the appropriate routing. If the statement is a conditional statement, then the variable in charge of keeping track of consecutive loops that have taken place, gets incremented.

- Continuation Statement: If the counter used to signify the level of conditionals that we are currently at is greater than 0, when the next sentence is read in, it is checked to see if it has a continuation statement. A continuation statement, is a sentence that contains keywords that make it seem like this statement extends what was spoken of in the previous sentence. If this condition is met, the instruction is taken to be a part of the conditional statement that was previously found. If the condition is not met, then the conditional statement is over and the termination delimiter of ”end” is added to the instruction queue in order to signify this for future functions. The variable in charge of keeping track of consecutive loops that have taken place, then gets decremented.

The keywords can then be abstracted from the sentence using the designed dictionaries shown in Appendix A. This should then give an output of an array of key-terms for the sentence. From here, the term could be taken directly and used for the instruction creation, or they can first be parsed into macros.

4.2.2 Macros

The general premise of a macro instruction falls under the idea that any complex instruction is just a combination of multiple simple instructions. The goal of this is to allow for more natural instructions, that do not seem robotic and can also convey more with less words. Once the keywords are taken, NGRAMS are used to match the term arrays to possible macros that could be generated.

It can be seen in Table 4.1 that the macros can be used to greatly reduce the work of the user by inferring the meaning of complex instructions, and breaking them down into smaller, and more basic instructions. In this case, rather than the user having to communicate every single movement that must occur to get the result desired, he or she can directly reference the end result and the predefined macro can handle the extrapolation.

Sentence	Keywords	Macro Instructions
the car should avoid the wall	[”car”, ”avoid”, ”wall”]	[’pan’, ’camera’, ’left’] [’get’, ’left’, ’distance’] [’pan’, ’camera’, ’right’] [’get’, ’right’, ’distance’] [’pan’, ’camera’, ’center’] [’if’, ’left’, ’distance’, ’greater’, ’right’, ’distance’] [’car’, ’turn’, ’left’] [’otherwise’] [’car’, ’turn’, ’right’] [’end’]

Table 4.1: Macro Example 1

Sentence	Keywords	Macro Instructions
the car should turn left	[”car”, ”turn”, ”left”]	[”car”, ”turn”, ”left”]
if the car can turn left	[”if”, ”car”, ”can”, ”turn”, ”left”]	[”pan”, ”camera”, ”left”] [”get”, ”left”, ”distance”] [”pan”, ”camera”, ”center”] [”if”, ”left”, ”distance”, ”greater”, ”30”, ”inches”]

Table 4.2: Macro Example 2

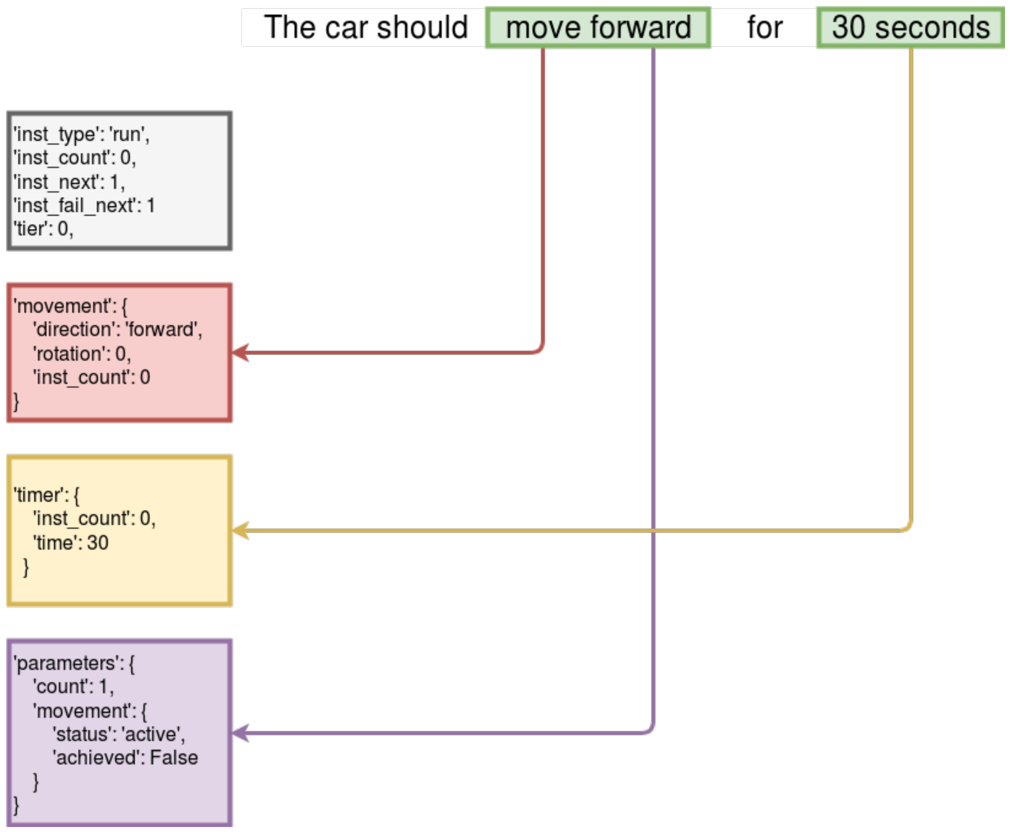
In the second example, Table 4.2, an even clearer view of the fluidity this allows for in natural language can be seen. The first row displays a simple command for the car to turn left. This is easily computed and broken down to its keywords to be made into the instruction. The only problem with this is that there is a general assumption that the rover is capable of completing this task, which may not always be true. Therefore, then second instruction shown takes just as many words to write down but is able to encapsulate more by adding a safety aspect to the entire scheme. This also removes the need for the user to know the specific boundaries (such as the turning radius of the car being 30 inches in this case) allowing them to focus in on other aspects of the system. Though, in both cases, if there became a safety issue during the running of the commands, there are still fail safes designed in the system, explained in the later chapters, it is important for the user to explicitly state the safety boundaries they feel necessary.

In the end, macros create a robust way of generating complex instructions through the use of simpler instructions. They also allow for simplicity in the way that instructions are crafted as well as in how much information the user must know beforehand.

4.3 Instruction Creation and routing

Once the key-terms have been obtained from the previous stage, a process like that used to generate the macros is used. In this, the structure and order of the keys are matched to n-grams that decide the type of instruction that needs to be generated. Depending on the type of instruction currently being written and the previous instructions that may relate to them, the instructions can then be routed accordingly.

To start off this process, the keyterms are analyzed in order to see what type of instruction they are. For example, if the statement contains the keyterm ”if” then the statement most likely routes to a CHECK statement. On the other hand, if it contains the keyterm ”while” or ”when” then this statement is most likely a LOOP statement. Depending on the type of instruction it is, cases are created to check for if the terms can be accommodated in the current syntax of the robot. If the sentence cannot be matched to a specific instruction, then an error is thrown.



```
'inst_type': 'run',
'inst_count': 0,
'inst_next': 1,
'inst_fail_next': 1
'tier': 0,
```

```
'movement': {
    'direction': 'forward',
    'rotation': 0,
    'inst_count': 0
}
```

```
'timer': {
    'inst_count': 0,
    'time': 30
  }
```

```
'parameters': {
    'count': 1,
    'movement': {
        'status': 'active',
        'achieved': False
    }
}
```

Figure 4.2: Instruction Creation Example

4.3.1 Basic Instructions

For the instructions that do not require conditions, all that needs to occur if for the text to be mapped directly into the instruction it is trying to represent. After this the instruction is placed inside the instruction array in the order that it is processed. An example of this can be seen in figure 4.2, the the sentence ”The car should move forward for 30 seconds” gets translated into its instructional form.

In this example it can be seen that the instruction is a direct breakdown of the parameters given in the statement. As no part of the statement relates to other instructions to come, there is no need for any other processing to occur after the instruction has been created.

4.3.2 Check Instructions

Like the basic instructions, check instructions use the N-gram key pairing method to create matches between the key words and the instruction desired. Due to this there is a direct match with the keyterms and the instructions that are generated. This can be seen in figure 4.3 in which the sentence ”If the left distance is greater than 30 inches” is translated into its instructional form. For this instruction, it can be seen that the keyword ”if” is used to classify this as a CHECK statement, while the other key words are used to establish the parameters that will be used in the comparison.

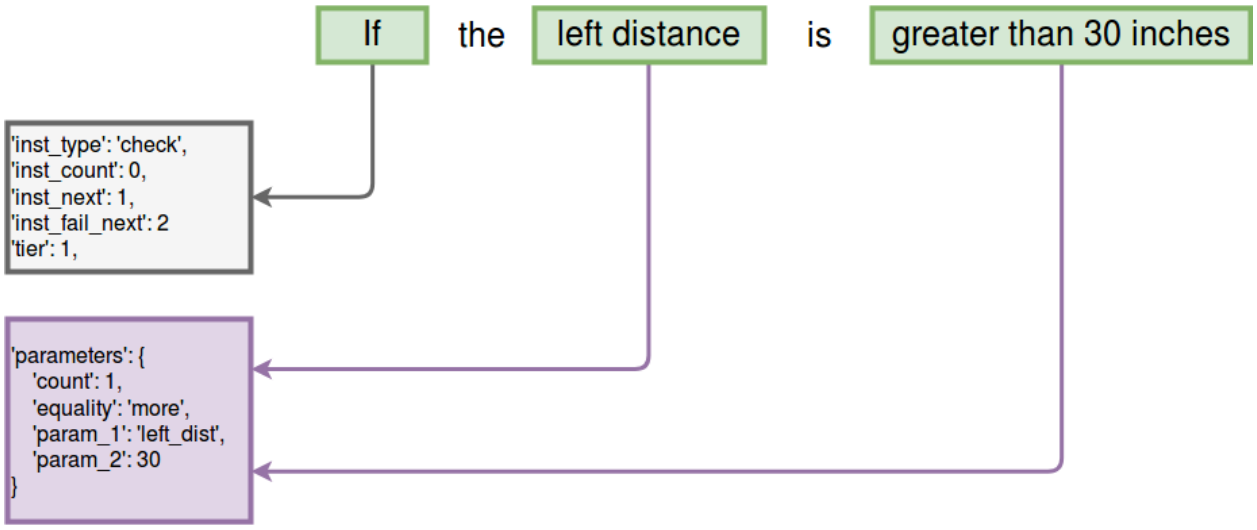


Figure 4.3: Check Instruction Creation Example

The complications that come with the CHECK statement instructions are when routing the statements from one to the other. As stated in the instruction types section, CHECK statements have three different variations based on how the statement needs to be configured.

With this, the normal ”check” instruction type is always expected to come before any of the other variations as this acts as the normal ”if” statement. The other two can then follow but the ”check_fin” acts like the ”else” statement, making it the end of the cases, meaning that the ”check_alt” cannot follow the ”check_fin”. Whenever a ”check” type instruction is encountered, the tier of the instruction is increased for easier routing. Any ”check_alt” or ”check_fin” instructions that follow are assumed to be in the same tier.

In the figure 4.4, an example of a complete check statement can be seen, as well as how it would be organized. In this example, the routing done by the different types of check statements is shown, and it can be seen how the statements are able to fluidly move and decide the best course of action given the different conditions that need to be met

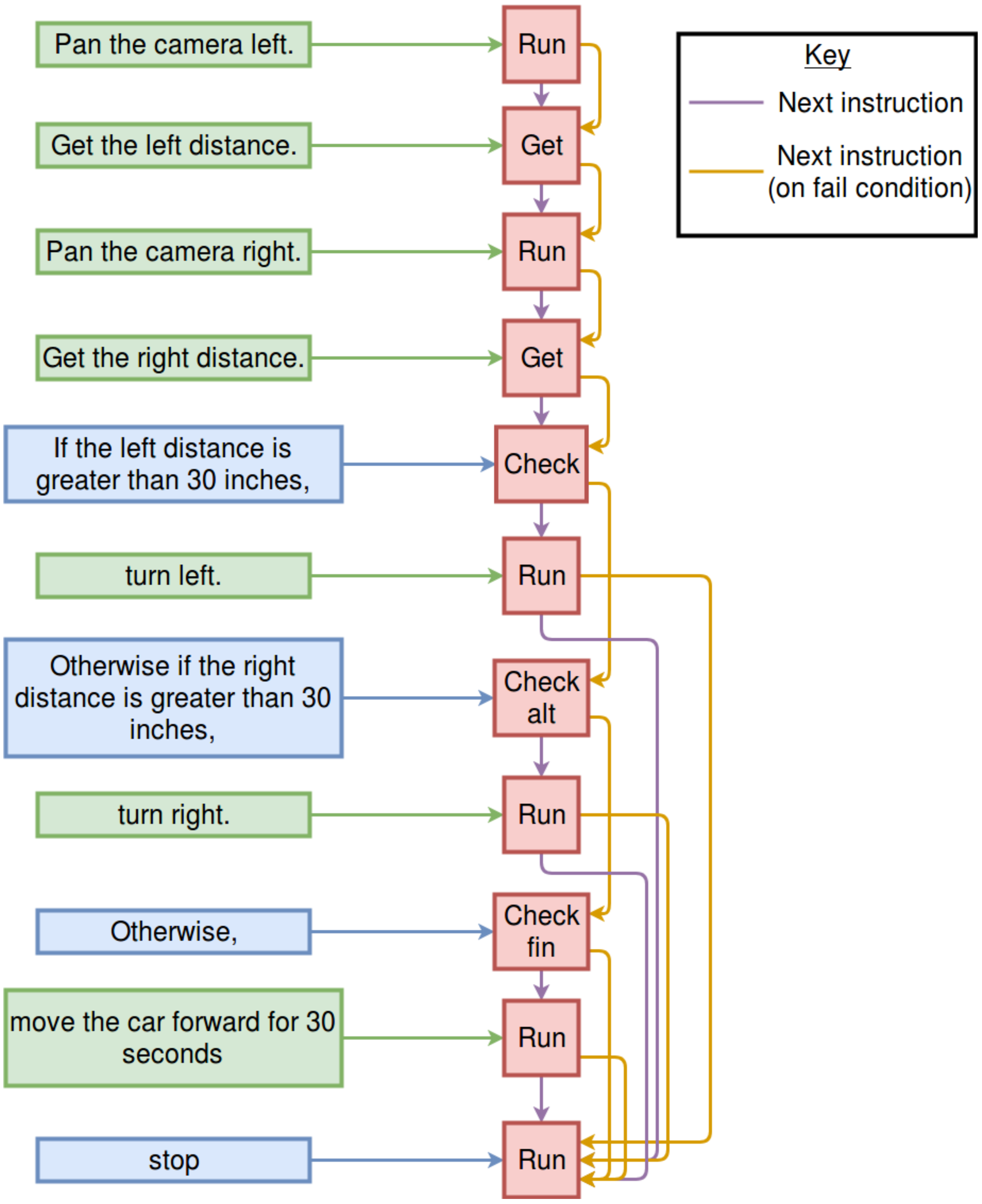


Figure 4.4: Check Instruction Routing Example

4.3.3 Looping Instructions

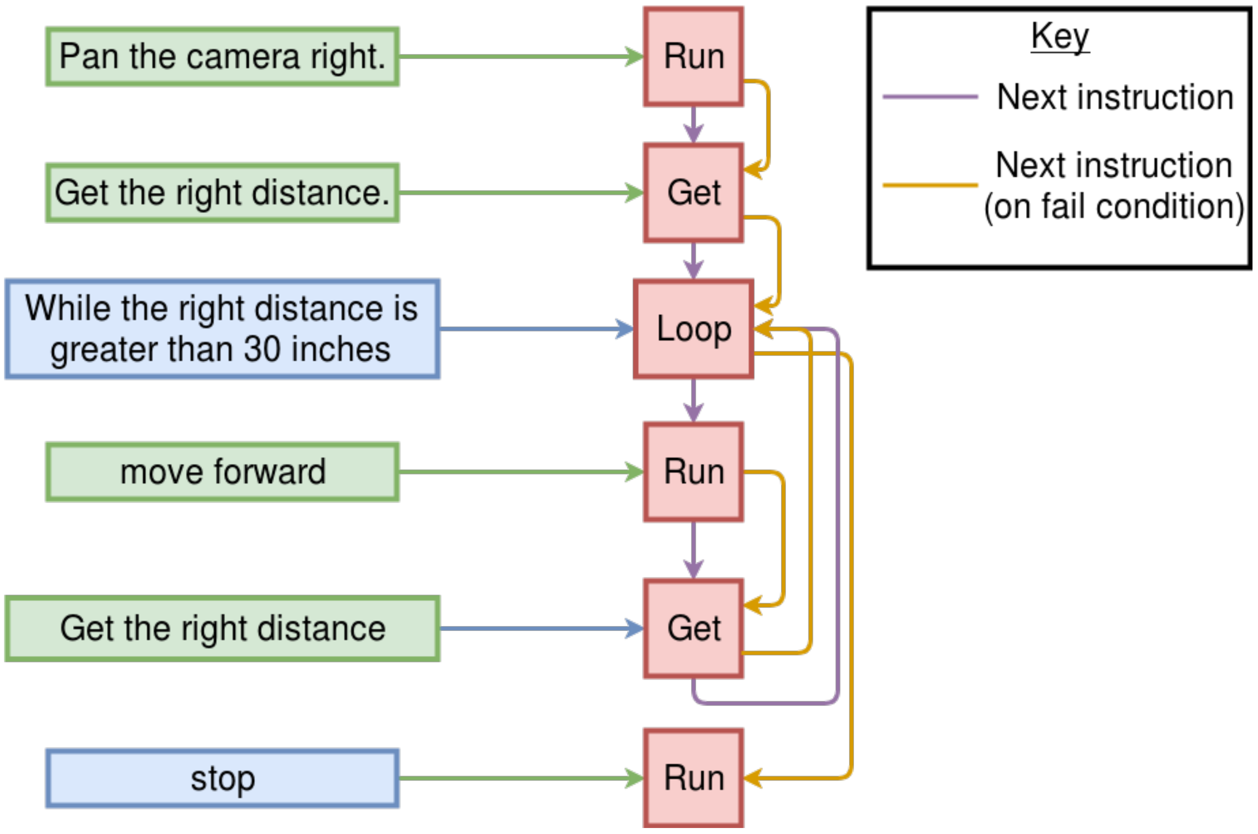


Figure 4.5: Check Instruction Routing Example

Like the CHECK statements discussed before, the complication that comes with the LOOP instructions takes place in how the instructions are routed at the end of the condition. The work in this case is made much simpler due to the fact that multiple alternative cases are not expected, like is assumed in the CHECK statement case. It is important to note that there are two types of LOOP instruction types, as stated in section 3. The first type, called the "loop_wait" instruction, does not bear much consideration in routing as it acts like any basic instruction, in that it does not take in other instructions to be looped through. Rather the instruction is looped in on itself until the condition is met. On the other hand, the "loop" instruction type can take in other instructions and loop through them until the condition is met.

In figure 4.5 an example of the loop routing can be seen. This example shows how the instructions withing the while loop are repeated until the condition is failed, then the final instruction is run.

4.3.4 Error Handling

There are multiple ways in which errors need to be looked at in this method compared to traditional programs that just look for syntactical errors. Some aspects that need to be taken in are the grammatical errors, errors in spelling and, of course, syntactical errors.

Grammatical Errors

When looking at the structure of the sentences, grammar becomes very important as it helps drive and define the order in which actions occur. Because of this, work needs to be done in order to check for grammatical problems, and these are the most common ones that are checked for:

1. **Comma placement:** As stated in the sections for the conditional statements, efforts need to be made by the instruction writer in order to separate each action within the conditional by commas. Due to this, errors are thrown if too many actions are not separated from one another.
2. **Key match errors:** If the sentence does not match any of the formats that are allowed my the syntax.
3. **Missing information:** Keys are appropriate but there is some missing information to make the sentence complete.

Chapter 5

ROS Nodal Structure

5.1 Overview

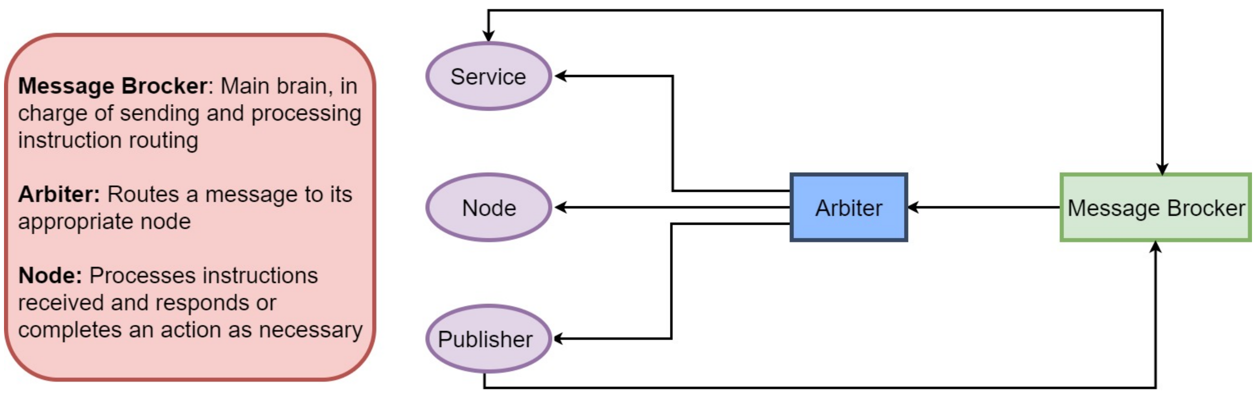

Message Brocker: Main brain, in charge of sending and processing instruction routing
Arbiter: Routes a message to its appropriate node
Node: Processes instructions received and responds or completes an action as necessary
Service
Node
Publisher
Arbiter
Message Brocker

Figure 5.1: ROS Routing Diagram

Due to the networking structure of ROS, there is an abundance of methods that can be used in order achieve the goals necessary for this research. Because of this, I decided to focus on a structure that would have a central control unit with several smaller units (nodes) to act as inputs and outputs for the system. With this in mind, the system shown in figure 5.1

was used as the main design. In this system, the central control and processing unit is the Message Broker node, with the Arbiter node acting as a messenger to the other peripheral nodes. Each node acts as a subscriber and/or a publisher in order to push information around with some nodes even having specific services attached to them. In the following sections i will describe the hardware used to implement this system as well as the different topics and services used by each of the nodes in order to facilitate this communication.

5.2 Hardware

For this system, both hardware and software simulations were used in the testing. Using ROS, it is then very easy to transfer functionality from one type to the other, by just changing a few lines within the nodes. In order to make this as seamless as possible, the simulated rover was designed based off of the actual physical rover. The main difference to note is that the physical system used an Ackerman steering style while the simulated rover used a differential steering style. this discrepancy was fixed in the software to ensure that the two systems had a similar turning radius. With that in mind, both rovers had the following characteristics:

- 4x wheels (two front and two back)
- 1x pan and tilt module
- 1x camera attached to the pan and tilt module
- 3x ultrasonic sensors (one attached to the front, one to the back, and the last on the pan tilt module)
- 1x IMU with 6 degrees of freedom

The physical system is running off of a raspberry pi 3, running Ubuntu Mate with a Teensy 3.2 attached for some low level sensor computing. This system also uses a regulated power distribution board as well as a dedicated I2C to PWM board in order not to damage the pins on the raspberry pi. The simulated system is running off of a Intel core I7 laptop with a dedicated Nvidia geforce 635M graphics card and 12 GB of RAM. The simulations are being run in Gazebo. Both of these can be seen in figures 5.2 and 5.3.

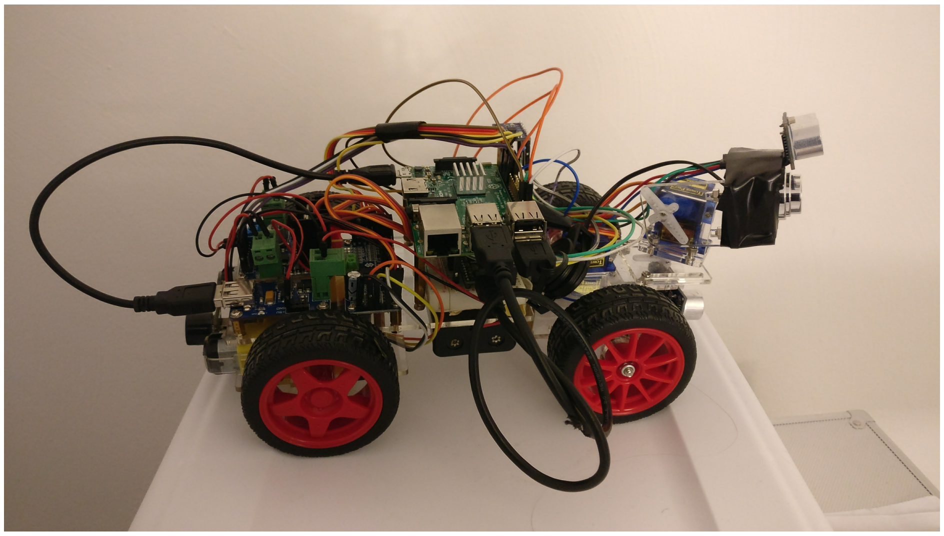


Figure 5.2: Physical car

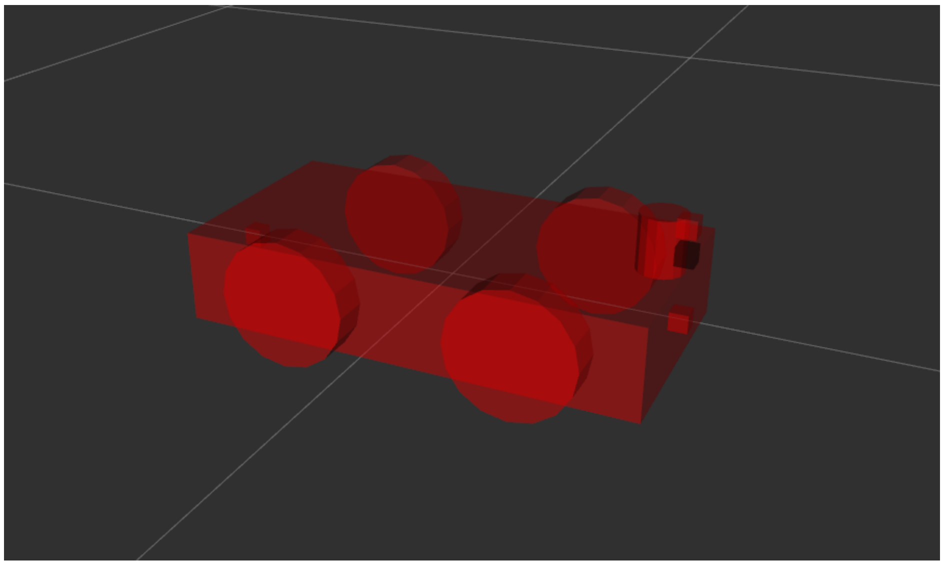


Figure 5.3: Simulated car

5.3 Network Structure

In designing the network for this system in ROS, nodes are at the core of what makes the system work. Remembering that the main goal of this system is to be able to break complex instructions up into multiple different smaller instructions, the ideology had to follow through on the ROS end as well. Because of this, nodes where created that had specific functionality, and that could be used in a simple and universal way. The idea around this was to have a central processing node, in charge of the overall logic and instruction handling, then have multiple smaller nodes in charge of the different aspects of the rover, such as movement and timing.

The nodes that are used in the project are the following:

- **Message Broker:**

- **Node Type:** Subscriber, Publisher, Client
- **Topics:**

  Subscribes to the ”sensors” topic

  Publishes to the ”instruction” and ”status” topics
- **Services:**

  Client for the ”get_movement_data”, ”set_movement_data” and ”get_ultrasonic_data” services
- **Description:** This is the main brain of the robot, in charge of handling all of the instructions. This node receives the instructions that have been sent in and begins to send the instructions to the arbiter nodes. As the tasks get completed, the completion messages are sent back to this node, and are used to base the next instruction that will be run. This node is also in charge to taking care of any errors that could occur while the hardware is running, such as collision detection.

• **Arbiter:**

- **Node Type:** Subscriber, Publisher
- **Topics:**

  Subscribes to the ”instruction” and ”status” topics

  Publishes to the ”movement”, ”timer”, ”ultrasonic” and ”pan” topics
- **Services:** N/A
- **Description:** The only goal of the arbiter node is to take in the instructions sent over by the message broker and send them out to the appropriate nodes that they should go to. Also, in the case that a system stop message has been sent out, this node is in charge of ensuring that the processes in all the nodes are stopped.

- **Movement:**
  - **Node Type:** Subscriber, Publisher, Service
  - **Topics:**

    Subscribes to the ”movement” topic

    Publishes to the ”sensors” topic
  - **Services:**

    Service for the ”get_movement_data” and ”set_movement_data” services
  - **Description:** The purpose of the movement node is to take care of all the motor movements of the rover. This node is in charge of the lateral movements (forward and backward) as well as any sort of turning that would need to be done. Sensor measurements are also taken in order to ensure that the vehicle is moving in a straight path.
- **Pan:**
  - **Node Type:** Subscriber, Publisher
  - **Description:** This node is in charge of the pan/tilt module attached to the rover. Attached to this module is an ultrasonic sensor, as well as the camera. This node is in charge of moving the pan/tilt module by rotating it to its appropriate rotation degree.
- **Ultrasonic:**
  - **Node Type:** Subscriber, Publisher, Service
  - **Topics:**

    Subscribes to the ”ultrasonic” topic

    Publishes to the ”sensors” topic

  - **Services:**

    Service for the ”get_ultrasonic_data” service

  - **Description:** This node is used to take the measurements of the three ultrasonic sensors. The measurements for the front and back sensors are published regularly, once the system has been started, while the measurements of the pan ultrasonic must be read in through a service call.

- **Timer:**

  - **Node Type:** Subscriber, Publisher

  - **Topics:**

    Subscribes to the ”timer” topic

    Publishes to the ”sensors” topic

  - **Services:** N/A

  - **Description:** The main goal of this node is to handle all the precise timing that could occur in the instruction. Each time a message is is received by this node an instance of the timer is created that runs for the period of time desired and publishes a response back once completed.

- **Ball Tracking:**

  - **Node Type:** Subscriber, Publisher

  - **Topics:**

    Subscribes to the ”/camera1/image_raw” topic

    Publishes to the ”sensors” topic

  - **Services:** N/A

  - **Description:** This node handles the image processing for the rover. Through the use of OpenCV, this node takes in the raw image data being sent by the camera and processes that do see if any obstacles can be detected in the line of sight. A message is then sent back through the sensors topic as to whether or not that object has been detected.

Communication had to occur between the different nodes in order to allow for the instructions and information to flow between them. For this, a mixture of topics and services were used depending on the type of messaging format that was necessary, with topics being created for nodes meant to run in the background and pump continuous information while other tasks were going on, and services being used for tasks that should be completed before other tasks/instructions can begin.

This is the list of all the topics and services used:

- **Instruction:**
  - **Type:** Topic
  - **Description:** This topic holds the instructions that need to be sent out to the different nodes.
- **Status:**
  - **Type:** Topic
  - **Description:** Used to update the status of the system with the main goal being to send system wide stop and start messages. This is necessary for particular nodes if they are sending out continuous information, like the ultrasonic node.
- **sensors:**

- **Type:** Topic
- **Description:** The main purpose of this topic is to store all the messages sent in by the different sensors. Each message sent to this topic contains the information on which sensor it came from so a separation of topics is unnecessary.

- **Response:**
  - **Type:** Topic
  - **Description:** This topic holds the instructions that need to be sent out to the different nodes.
- **Movement:**
  - **Type:** Topic
  - **Description:** In charge of all the information going to the movement node.
- **Pan:**
  - **Type:** Topic
  - **Description:** In charge of all the information going to the pan node.
- **Ultrasonic:**
  - **Type:** Topic
  - **Description:** In charge of all the information going to the ultrasonic node.
- **Timer:**
  - **Type:** Topic
  - **Description:** In charge of all the information going to the timer node.

- **get_movement_data:**
  - **Type:** Service
  - **Description:** Used to get data from the movement node, like the current action the rover is trying to take.
- **set_movement_data:**
  - **Type:** Topic
  - **Description:** Used to set data in the movement node, like the speed of movement.
- **get_ultrasonic_data:**
  - **Type:** Service
  - **Description:** Used to get data from the ultrasonic node. Mainly to get sensor data for the panning ultrasonic.

Chapter 6

Results

6.1 Overview

There is no universal mechanism for testing natural language programming since, as stated before, the experience is driven highly by the specific situation it is being designed for. Due to this, the results will be broken down into different cases, with each explaining the problem that needs to be solved and how that was possible using the instructions built with the natural language processor. In each of these cases, a visual representation of the node routing will be shown. This visual tree diagram is used to show the complexity of the instruction as well as to ensure that the routing is made as expected. Figure 6.1 can be used as a key to know what the different arrows and numbers on the diagram mean.

As a review, the main challenges that are being looked at through these tests are as follows:

- The ability to analyze text for any syntactical errors.
- The ability to convert the text into instructions for both basic and conditional statements.

- The ability to chain these instructions in a logical way as to allow them to flow seamlessly between each other.
- Develop a structure within ROS that can take in these instructions and run them.
- Allow for some security in terms of collision detection for the rover to handle autonomously.

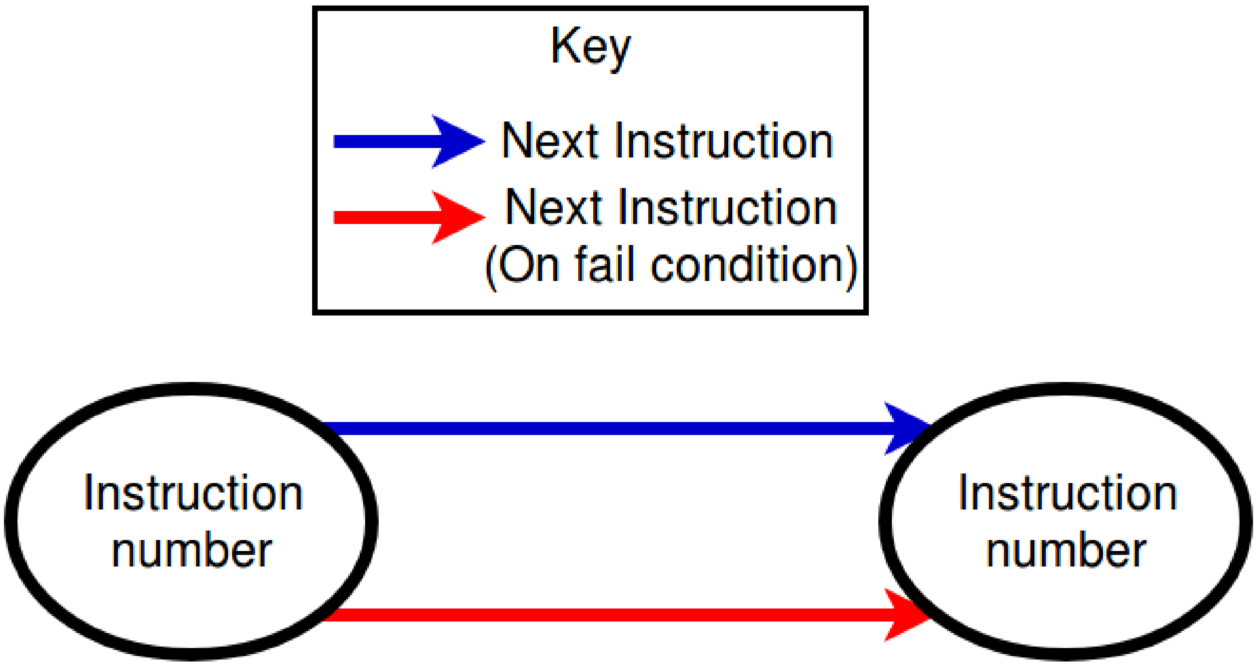


Figure 6.1: Instruction Tree Key

We will begin by looking at how the safety features that are built into the system worked as well as how the basic instructions and complex instructions ran. Then the first case that will be looked at will be a simple object detection case, testing the camera instructions. After this I will look into more complex cases as the rover tries to traverse two different mazes.

6.2 Collision and Error Detection

Errors in the text were caught as expected, throwing an error and indicating the possible problem that could have caused the error. This was mainly due to the use of the N-gram structure matching, that looked to ensure that the sentence fell within a specific structure and anything outside of that structure was subject for correction. In cases where close matches were seen, in that the structure followed one of the specific formats but may have added a few extra details, the closest match was taken. This approach seems satisfactory as human sentences are not an exactly the same from person to person, so this allows for the variability between users.

In most of the cases tested for the collision detection aspects, the rover was able to come to a complete stop when it sensed that an imminent collision was about to occur. The only cases when this did not happen was when the car was moving slightly sideways and scraped against the wall. This problem is mainly attributed to the design of the car since it only has a single ultrasonic sensor to detect distance in the left and right direction. It is important to look at the safety features of this in order to analyze its feasibility in a real life scenario. Currently, the design of the vehicle itself would have to be readjusted to fit actual safety requirements.

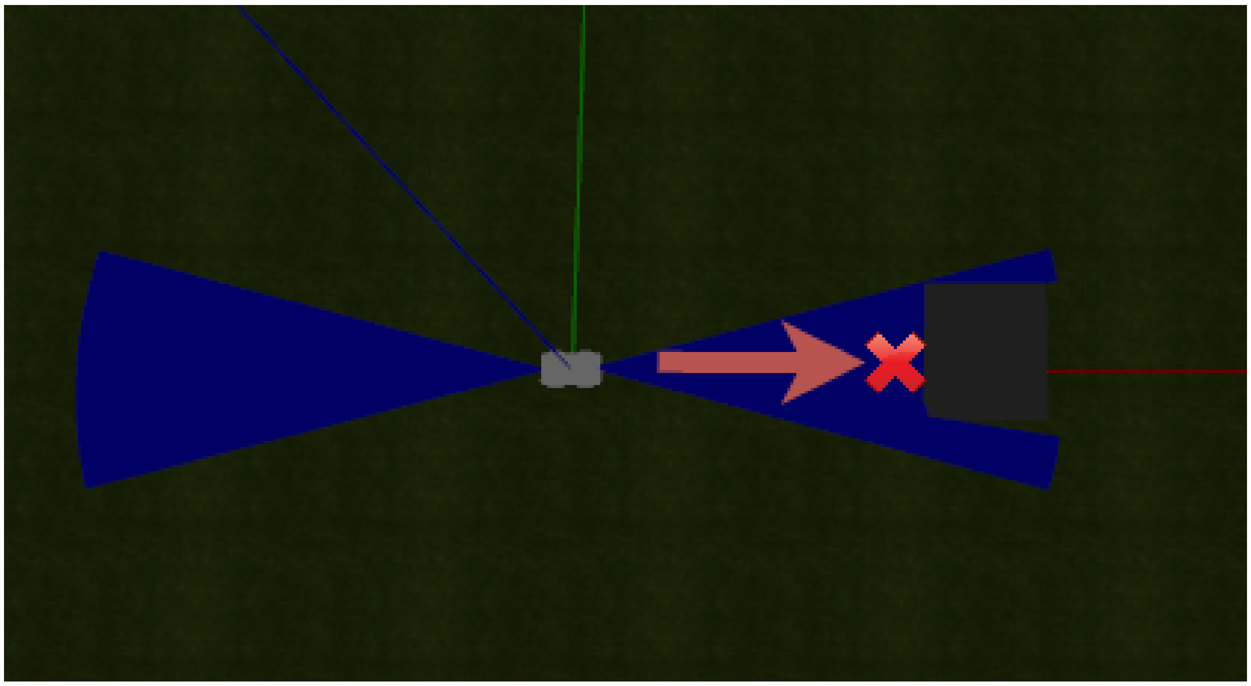


Figure 6.2: Collision detection example

6.3 Basic Instructions

In the case of the basic instructions, they all acted as expected. As there is no complication in their routing, they have a very simple tree as shown in figure 6.4. Also due to its simplicity and nature, the basic instructions had a one-to-one correlation with the number of instructions that got generated, as shown in the table 6.1. Overall these instructions were simple enough to work in all the cases they were tested in.

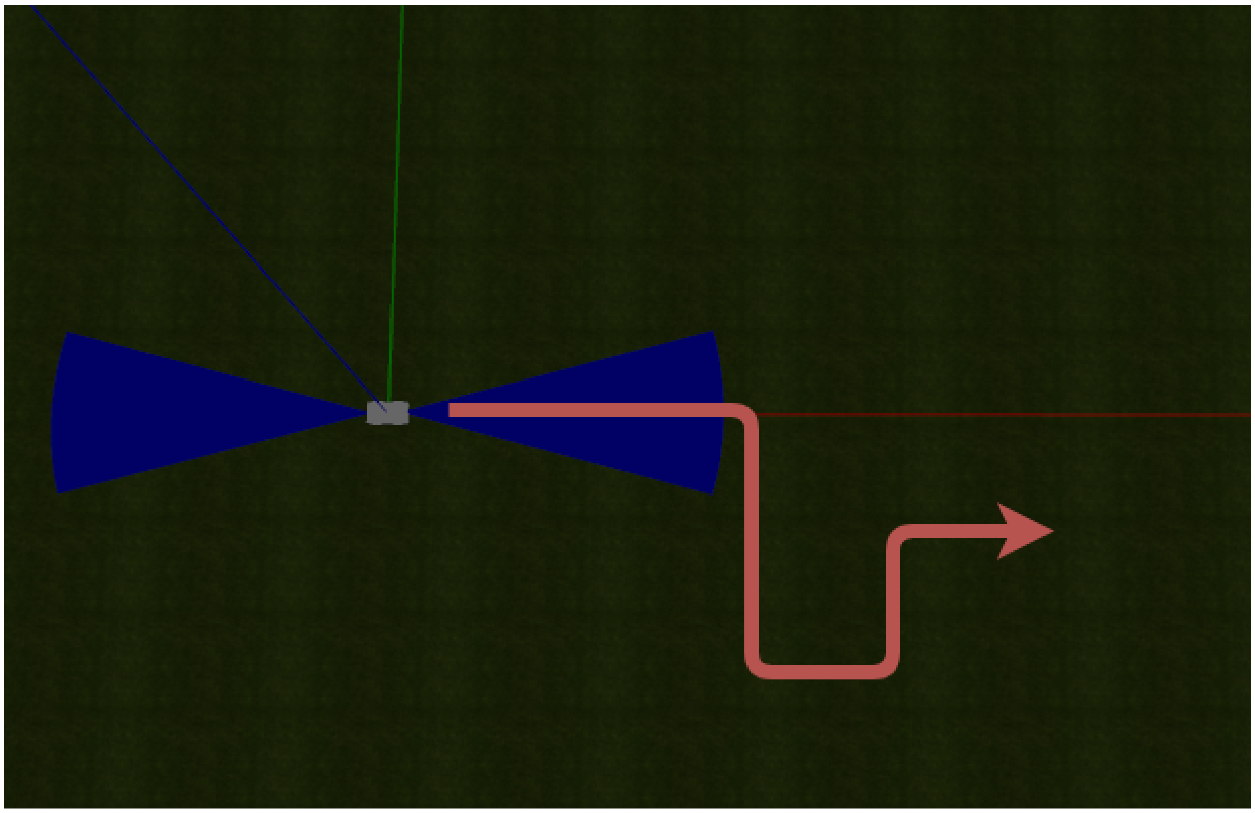


Figure 6.3: Basic instruction example

Basic Instruction Example	
Text Input	Instructions Generated
the car should move forward for 2 seconds	8
turn right	
move forward for 2 seconds	
turn left	
turn left	
turn right	

Table 6.1: Basic Instructions Example

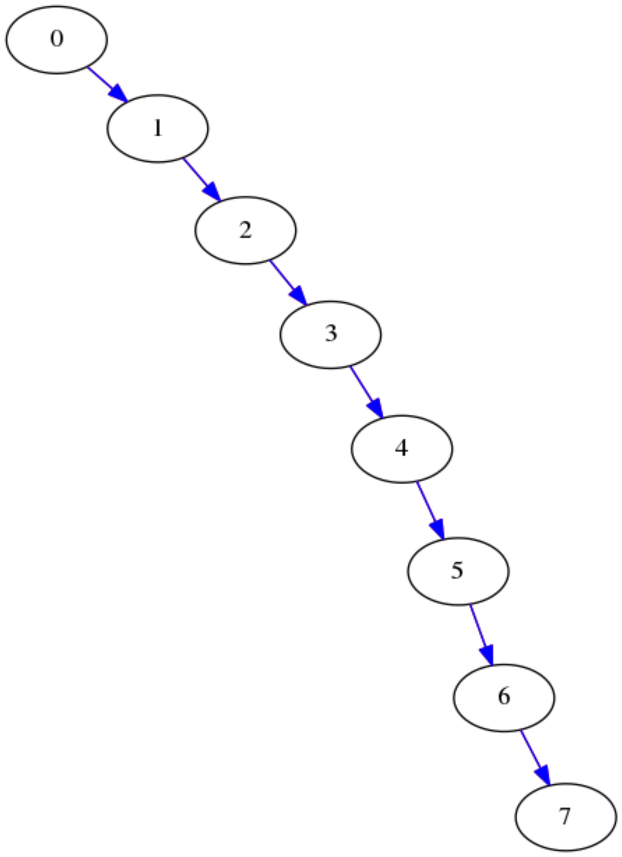


Figure 6.4: Basic instruction tree

6.4 Complex Instructions

6.4.1 Basic Usage

In the basic usage for the conditional instructions, it can be seen that they share a direct correlation with the instructions generated, again having that one-to-one ratio like the basic instructions. In this example, though, macros are used to do some extra work in checking the environment. Before any checks can occur, the environment needs to be scanned and this is shown by the first six node in the tree shown in figure 6.6. After this, it can be seen how the instructions are cascaded in a way that allows the conditions to adjust the outcome based on the different inputs given. In the end, each of the paths then come back to the last node. With this, the functions acted as expected, allowing for the CHECK conditionals to be presented as they should.

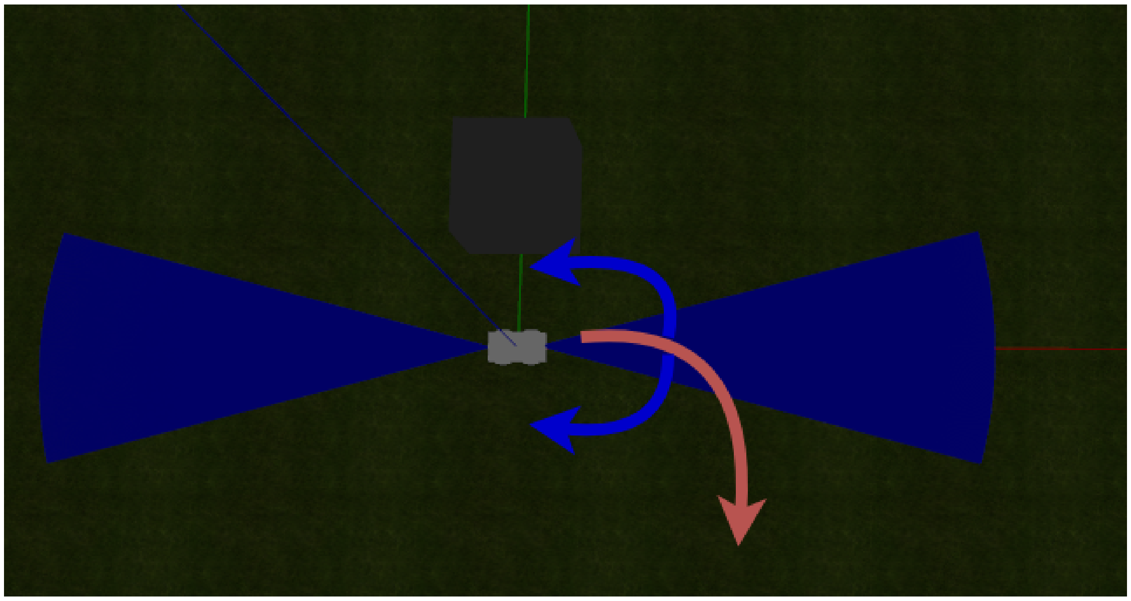


Figure 6.5: Complex instruction example 1

Example 1 Instructions	
Text Input	Instructions Generated
if the car can turn left, turn the car left otherwise if the car can turn right, turn the car right otherwise, move forward for 2 seconds	14

Table 6.2: Complex Instructions Example 1 Instructions

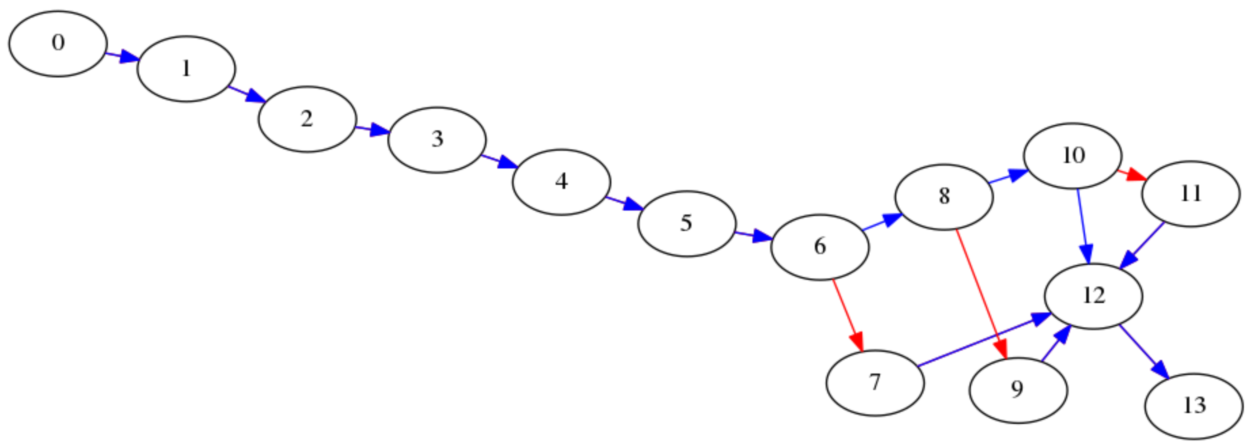


Figure 6.6: Complex instruction tree 1

6.4.2 Use Case

In this case, a more realistic example is given as the rover attempts to move around an obstacle. A very direct approach is used in which each of the steps that are expected to be taken are written out and no macros are used. In the actual case examples later on, the benefits and drawbacks of this are listed, but here it serves the purpose of viewing how the text relates to the instructions generated. The direct correlation between the sentence and the instruction generated can more easily be seen than before, as each sentence, more or less, is tied to its own specific instruction. Looking at the tree in figure 6.8, it can be seen where the thee looping instructions are located (nodes 0, 5, and 11). This shows how easily this design structure can be modified in order to account for different conditions that may occur.

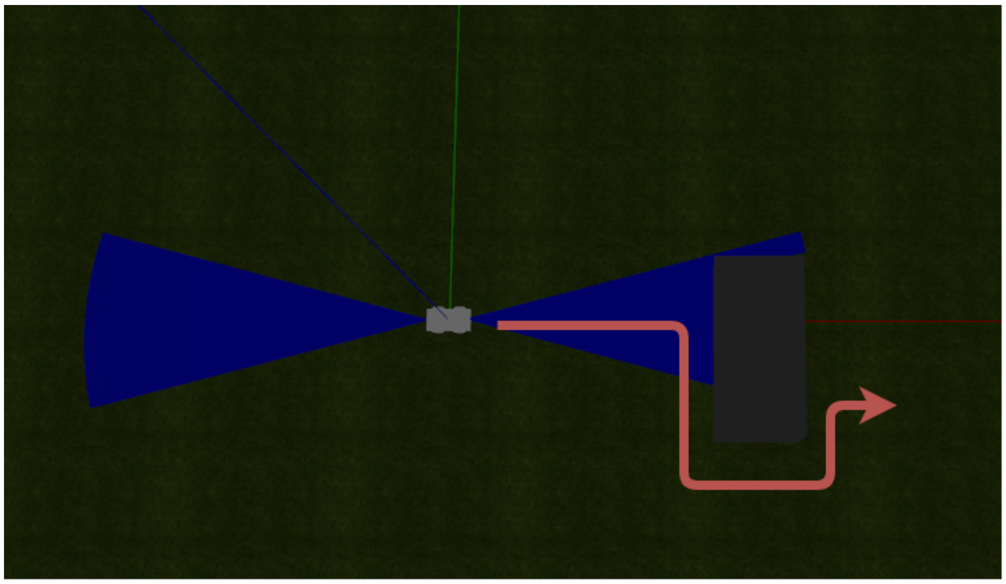


Figure 6.7: Complex instruction example 2

Example 2 Instructions	
Text Input	Instructions Generated
while the car is greater than 40 units from the wall, move forward turn right pan the camera left get the left distance while the left distance is less than 40 units, get the left distance, move the car forward the car should turn left move forward for 3 seconds get the left distance while the left distance is less than 40 units, get the left distance, move the car forward the car should turn left turn right pan the camera to the center	19

Table 6.3: Complex Instructions Example 2 Instructions

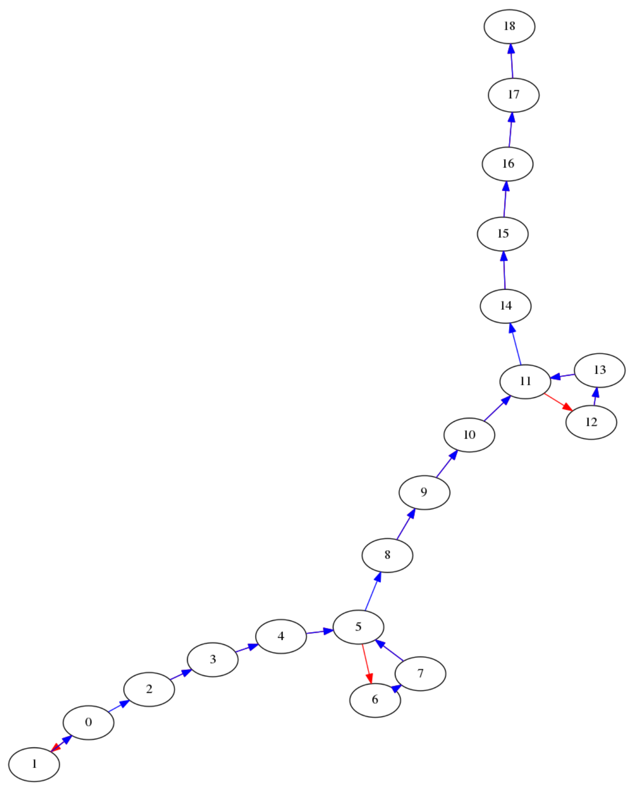


Figure 6.8: Complex instruction tree 2

6.5 Case 1: Object detection

In this test case, goal goal was to check for accurate and real-time object detection instructions in a simple case. As stated earlier, object detection is not a main focus of this research, but it represents a useful basis for testing. The car begins pointed away from the ball, and is meant to rotate until it detects the ball, as shown in figure 6.9. The instructions given for this case are shown in table 6.4, and in this same table it shows that there are 5 instructions generated from this two part sentence.

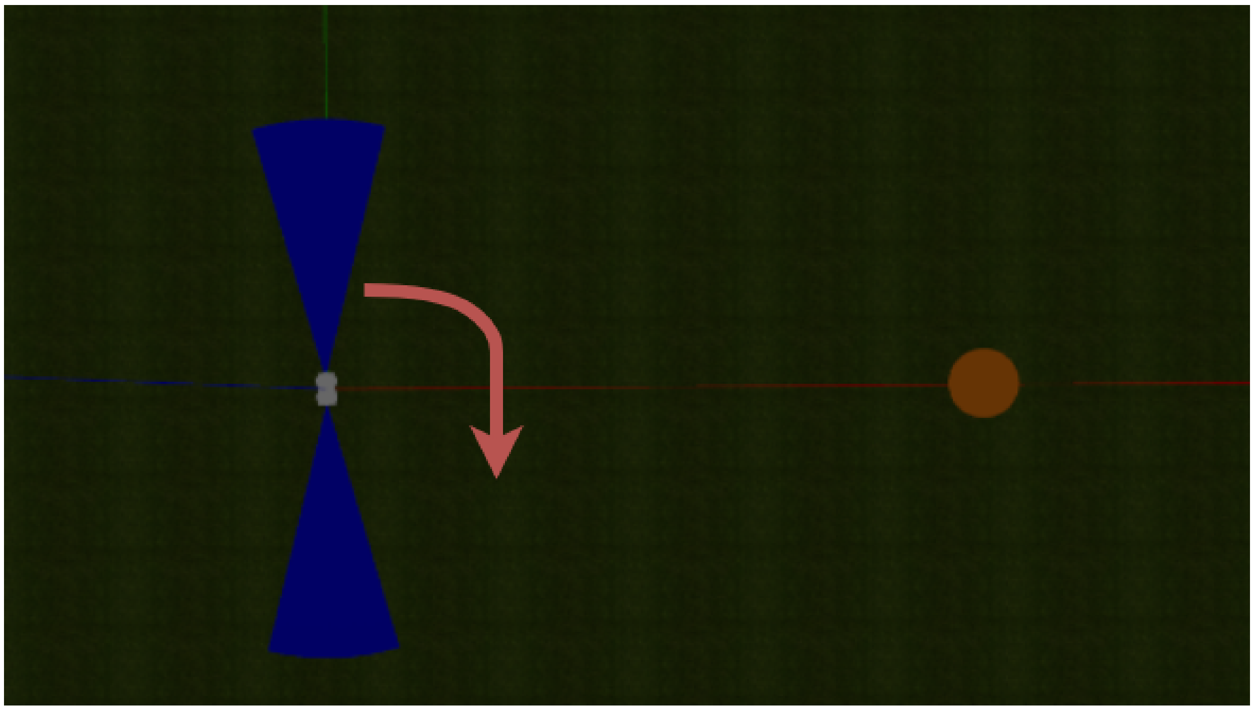


Figure 6.9: Object Detection Path for World 1

Wold 1 Object Tracking	
Text Input	Instructions Generated
turn the car left. while the camera cannot see the orange ball, turn the car right 5 degrees.	5

Table 6.4: World 1 Object Tracking Instructions

The code for this runs and generates as expected, with the instruction tree being shown in figure 6.10. An analysis of this tree would show that instruction 0 does the first movement to get the original orientation, followed by instruction 1 which does the continuous check to see if the object has been detected. If it has not been detected, instruction 2 is called, rotating the car 5 degrees. This continues until the ball has been detected and the final instruction stops the rover. The structure of this is important to note, because the obstacle detection node is never called outright in any of the instructions, like the other nodes commonly are, but rather, updates the main message broker node directly when new information is gained. This was done mainly to test how this system would work comparatively to the other systems used and it seems to work just as well.

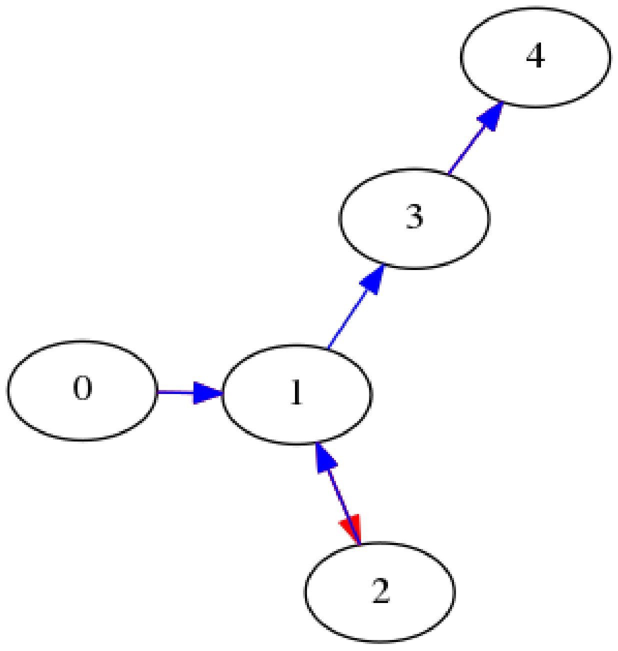


Figure 6.10: Object Detection Tree for World 1

6.6 Case 2: Maze 1

For this case, I show three different methods that can be used to obtain the same result to solve the first maze. For this maze, the rover must simply move forward until it gets to the corner, then make a left turn, move forward to the next corner, and then make a right turn. As with all the cases, the assumption is made that the end user knows the track and what actions need to be done, and for this, different variations on the idea are used to show the varying complexities that can be taken when producing the instructions that need to be taken.

6.6.1 Path 1

In this first case, the instructions are given directly by the user as to how the movement should be done and when the turns have to be executed. In this, the simplest and most direct case, the user tries to track the right wall using the wall follow macro at its default settings, then constantly checks the forward distance to tell when the rover has neared the wall. When this happens, the rover turns left, and follows the same procedure into the next corner and turns right. The main benefit that can be seen from this is that the structure is very direct, with the user constantly knowing how the rover will react at each instant and it does not require too much thought on keeping track of the sensors or values being passed. The main problem, then, comes in from the fact that the instructions are too simple to allow for smooth movement, making the rover move in a path that follows the default values of the macros. This allows for inconsistencies between runs, and for poor movement as can be seen in figure 6.11.

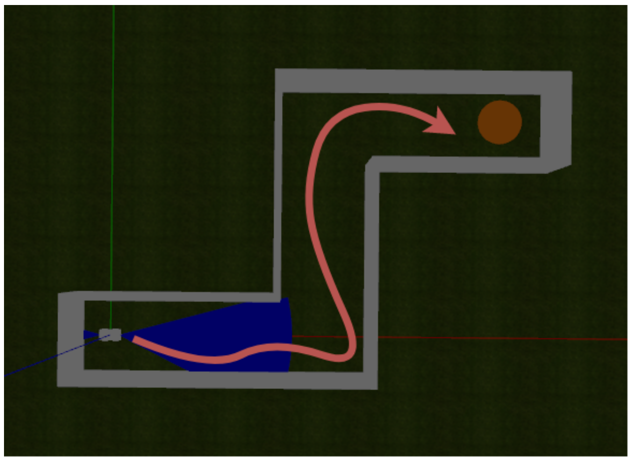


Figure 6.11: Object Detection Path 1 for Maze 1

Maze 1 Path 1	
Text Input	Instructions Generated
pan the camera right	36
get the right distance	
while the distance to the wall is more than 30 inches, follow the right wall	
turn the car left	
get the right distance	
while the right distance is less than 50 inches, follow the right wall	
turn the car right	
pan the camera left	
get the left distance	

Table 6.5: Maze 1 Path 1 Instructions

Another problem is also the repetitive nature of the actions being taken, and this is most easily seen in the tree in figure 6.12. In this, it can be seen that there are three different clusters of instructions, centered at nodes 2, 13, and 25, with all of them following the same idea of checking to see if the corner has been detected. This is very inefficient as it generates more instructions than necessary.

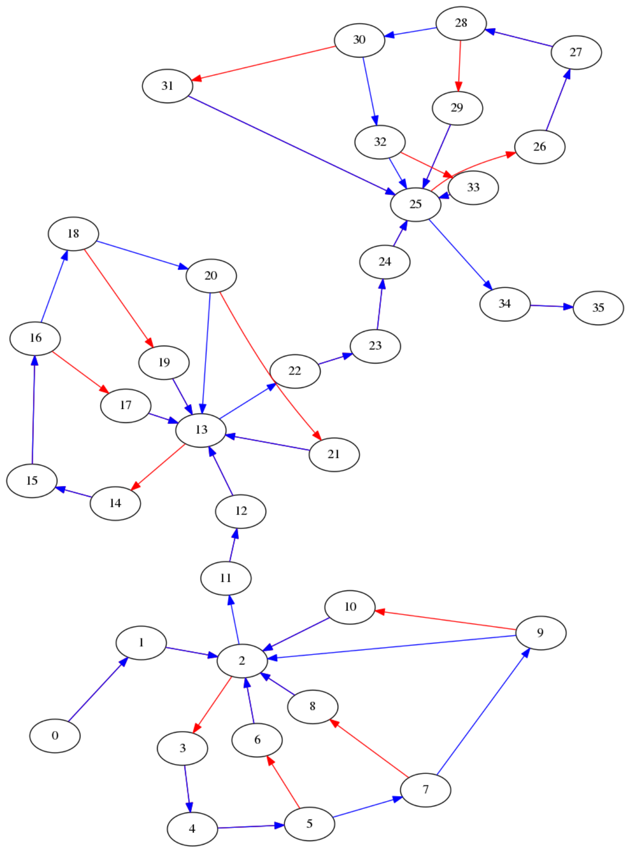


Figure 6.12: Object Detection Tree 1 for Maze 1

6.6.2 Path 2

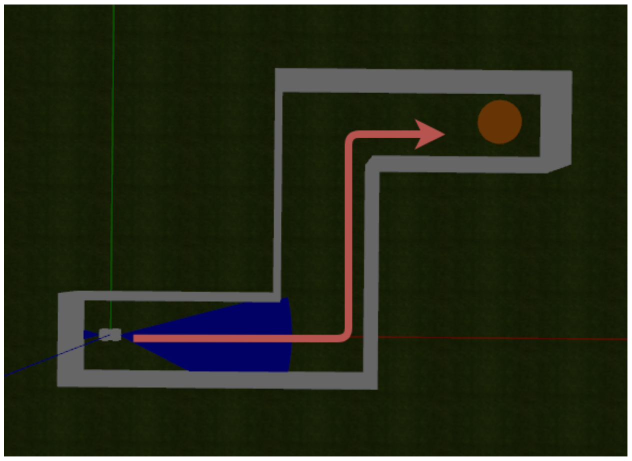


Figure 6.13: Object Detection Path 2 for Maze 1

Maze 1 Path 2	
Text Input	Instructions Generated
pan the camera right get the right distance set the measurement distance to the right distance while the forward distance is greater than 30 inches, follow the right wall using the measurement value stop turn the car left get the right distance set the measurement distance to the right distance while the right distance is less than 50 inches, follow the right wall using the measurement value stop turn the car to the right pan the camera to the left get the left distance set the measurement value to the left distance while the distance to the wall is greater than 30 inches, follow the left wall using the measurement value stop	42

Table 6.6: Maze 1 Path 2 Instructions

For this second path, the same idea as before is used, of directly giving the instructions, but with the twist that more thought is placed into allowing for a smoother movement of the rover. This is done by the user constantly setting internal values to be used to judge distance against, rather than using the default values of the macro. The allowance of this sort of setting and getting of values makes for smoother movement and more consistent results between runs as can be seen in figure 6.13. Table 6.6 does show a problem with this in that the user must constantly keep track of the measurement value at each turn and ensure that it is being updated accordingly. This path still, as shown in figure 6.14, does not fix the problems with the inefficient creation of multiple repetitive instructions.

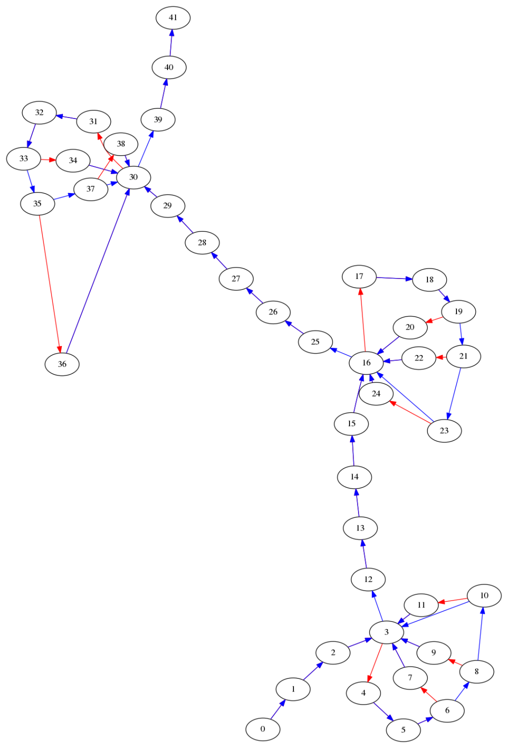


Figure 6.14: Object Detection Tree 2 for Maze 1

6.6.3 Path 3

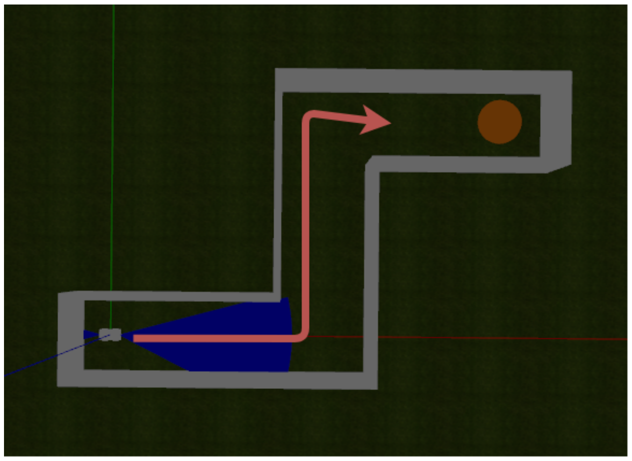


Figure 6.15: Object Detection Path 3 for Maze 1

Maze 1 Path 3	
Text Input	Instructions Generated
pan the camera left get the left distance set the measurement value to the left distance while the camera can not see the orange ball, follow the maze	27

Table 6.7: Maze 1 Path 3 Instructions

In this final case, a macro is used in order to encapsulate the instructions necessary to follow the maze. With this, it can be seen that the smooth flow of the rover is the same as before in figure 6.15, and the number of instructions generated is much less in table 6.7. Looking at figure 6.16, it can be seen that this is due to the fact that all the repetitive instructions have been reduced, creating a nice and circular flow to the system. The main problem that then comes with this is that most of what goes on in the macro acts like a black box, with the user having not as much direct control of the actions that will be taken by the rover. As shown in the next section, this may give consistent results for less code, it does not adjust as well to different scenarios. Another problem with this sort of method is having to know exactly what other parameters must be taken into account when wanting to adjust the different internal variables.

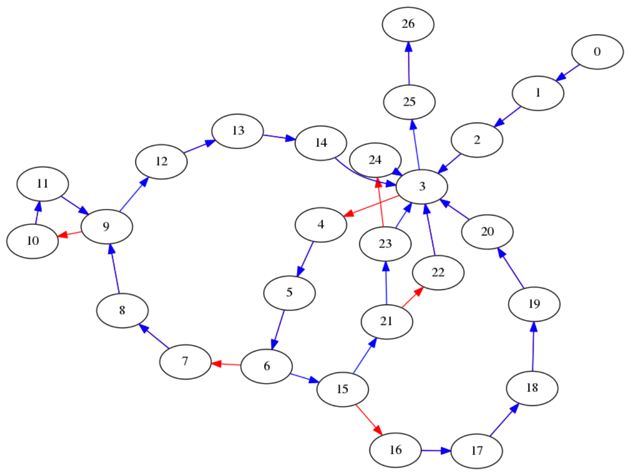


Figure 6.16: Object Detection Tree 3 for Maze 1

6.7 Case 3: Maze 2

For this next maze, the rover must travel across the maze and make a left in order to get to the ball. The main challenge then comes in the fact that there are now multiple paths that can be taken to reach this same goal. With this in mind, two different cases are shown that try to accomplish this, with each having its own benefits and drawbacks.

6.7.1 Path 1

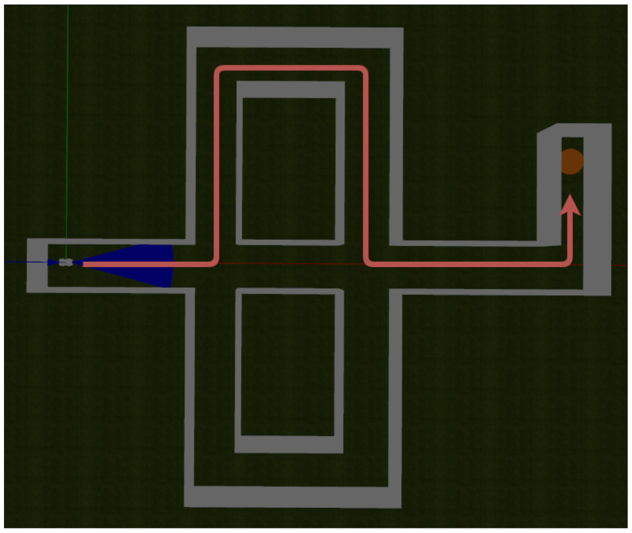


Figure 6.17: Object Detection Path 1 for Maze 2

Maze 2 Path 1	
Text Input	Instructions Generated
pan the camera left get the left distance set the measurement value to the left distance while the camera can not see the orange ball, follow the maze	27

Table 6.8: Maze 2 Path 1 Instructions

In this first path, the same instructions were given as that at of the last path in the previous case. With this, like before, no repetitive tasks are generated, as seen in figure 6.18 making the number of instructions generated minimal, shown in table 6.8. Looking at figure 6.17, though, it can be seen that, though this path does eventually lead the rover to the ball, it does not take the most efficient path possible to do this. This goes back to what was discussed before in that macros can act as a black box, following a fixed algorithm, and only being customizable to a certain extent. They are definitely useful for simplifying what needs to be written but more thought needs to be put into the instructions to allow for better efficiency.

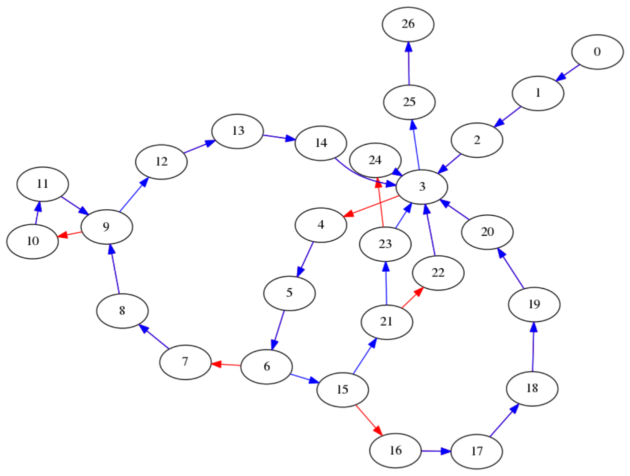


Figure 6.18: Object Detection Tree 1 for Maze 2

6.7.2 Path 2

In this next case, rather than using a macro to accomplish the whole task, the user breaks up the task into sections and acts according to what needs to be done for that particular section as shown in table 6.9 and figure 6.20, this does generate more instructions, therefore making the tree structure more complex, but figure 6.19 shows that the main benefit is that the rover moves in a more direct and efficient path. This shows that many aspects need to be looked into when trying to analyze the over efficiency of the code being generated and compared to the amount of learning that would need to be done on the users end to best utilize the system.

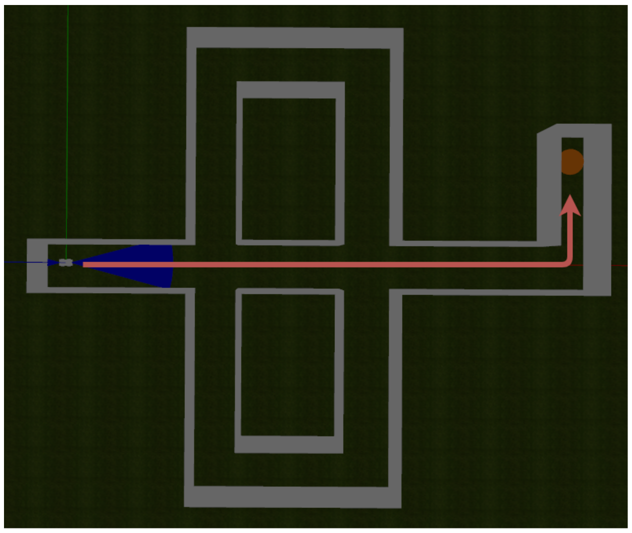


Figure 6.19: Object Detection Path 2 for Maze 2

Maze 2 Path 2	
Text Input	Instructions Generated
pan the camera left get the left distance set the measurement value to the left distance while the left distance is less than 50 inches, follow the left wall with the measurement value stop while the left distance is greater than 30 inches, get the left distance, move forward stop set the measurement value to the left distance while the left distance is less than 50 inches, follow the left wall with the measurement value stop while the left distance is greater than 30 inches, get the left distance, move forward stop set the measurement value to the left distance	57

Table 6.9: Maze 2 Path 2 Instructions

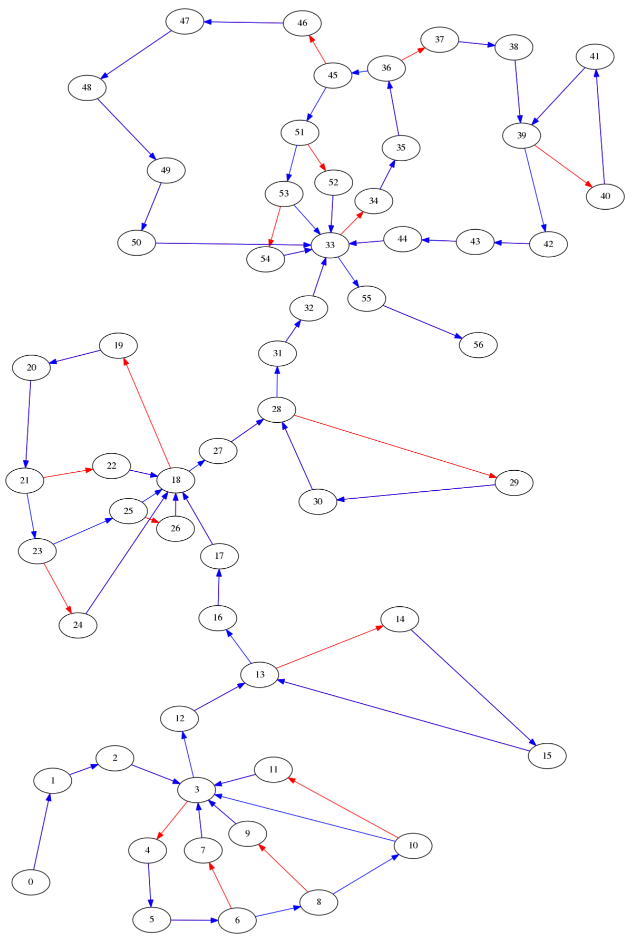


Figure 6.20: Object Detection Tree 2 for Maze 2

Chapter 7

Conclusion and Future Work

With the growing trend in autonomous control, it is important to take into account that, though autonomous systems can solve many problems on their own outright, there are still many cases in which humans and robots must interact directly in order to solve problems. Due to this, it is important to take note of the ways in which humans react with each other and form a basis on how this can be translated into a relationship with robots. For this reason, this research tries to allow for direct human-robot interaction in a way that not only allows for direct translation from natural language to instructions, but does it in a way that is robust and does not require too much learning from the user.

7.1 Conclusion

The main goal of this system was to take natural language text and transform it in a way that instructions could be generated to encapsulate the different actions that could be taken by a rover. The text for this was accurately parsed, checking for errors and inconsistencies, while at the same time being examined for extended capabilities of using a sentence to describe multiple actions. By using the idea of accomplishing complex actions through the use of simple actions, I was able to easily move past the challenge of taking in a complex task and being able to break it down into smaller tasks. The use of N-grams aided in allowing for syntax matching and allowed for easy translation for both the basic and complex instructions, solving the challenge of accounting for the conditional statements. The instruction completeness allowed for a variability of instruction types that could handle the different situations and structures of sentences that could be posed.

Looking at the results of the cases shown, there is a large amount of robustness that can be generated from the natural instructions. This robustness allows for various methods to be used to obtain the same goals and allows the user to choose the specific result that they would like to obtain. As was shown in those cases, the work load can change in complexity given how direct the user would like to be, defining just how many macros they would like to use at a time. This design complexity must be counteracted with the amount of information about the system that the user is willing to learn in order to account for the overall efficiency. All of these factors must then be judged against the scenario given in order to gain the best results. Looking at all these factors I believe that i can accurately state that I have accomplished the goals that were originally set and have demonstrated the contributions that I set out to create.

7.2 Future Work

Moving forward, there are many additions that can be added to make the system run more efficiently and be more robust to different scenarios:

- A more in depth look at the grammar and spelling detection and correction, as this was not focused on in this research but is essential when there is a human factor as the input.

- The introduction of the for loop type of instruction for adding timing constraints on a looping conditional statement.
- Allowing for multiple conditions to be set rather than just a single condition in order to allow for more robustness.
- The simplest of these is simply adding more peripherals to the system and trying to incorporate them in with the instruction types. This would allow for a more interesting robot by creating larger variability in the number of actions that it can take. One of these is allowing for interaction between the rover and other obstacles in the area, meaning that attachments need to be added to interact with these objects and more complex image processing would need to be used.
- Adjusting the code to allow for multiple rovers to be controlled at once. This could introduce the idea of the swarm style of robotics in which commands would need to be generated in order to ensure that the rovers do not collide and are moving in a way that is efficient for the group.

Bibliography

[1] Scott Neuman. ’flippy’ the fast food robot (sort of) mans the grill at caliburger. `https://www.npr.org/sections/thetwo-way/2018/03/05/590884388/flippy-the-fast-food-robot-sort-of-mans-the-grill-at-caliburger`, Mar 2018. Accessed: 2018-05-15.

[2] K. Ponmani and S. Sridharan. Human–robot interaction using three-dimensional gestures. In Daniel Thalmann, N. Subhashini, K. Mohanaprasad, and M S Bala Murugan, editors, *Intelligent Embedded Systems*, pages 67–76, Singapore, 2018. Springer Singapore.

[3] R. H. Taylor and D. Stoianovici. Medical robotics in computer-integrated surgery. *IEEE Transactions on Robotics and Automation*, 19(5):765–781, Oct 2003.

[4] Moustris G. P., Hiridis S. C., Deliparaschos K. M., and Konstantinidis K. M. Evolution of autonomous and semiautonomous robotic surgical systems: a review of the literature. *The International Journal of Medical Robotics and Computer Assisted Surgery*, 7(4):375–392.

[5] M. Huber, M. Rickert, A. Knoll, T. Brandt, and S. Glasauer. Human-robot interaction in handing-over tasks. In *RO-MAN 2008 - The 17th IEEE International Symposium on Robot and Human Interactive Communication*, pages 107–112, Aug 2008.

[6] William F. Gilreath and Phillip A. Laplante. Instruction set completeness. *Computer Architecture: A Minimalist Perspective*, page 5571, 2003.

[7] Nicholas K. Lincoln and Sandor M. Veres. Natural language programming of complex robotic bdi agents. *Journal of Intelligent & Robotic Systems*, 71(2):211–230, Aug 2013.

[8] P. Yin. Natural language programming based on knowledge. In *2010 International Conference on Artificial Intelligence and Computational Intelligence*, volume 2, pages 69–73, Oct 2010.

[9] M. C. Surabhi. Natural language processing future. In *2013 International Conference on Optical Imaging Sensor and Security (ICOSS)*, pages 1–3, July 2013.

[10] W. Eric L. Grimson and Ramesh S. Patil. *Intelligent Natural Language Processing: Current Trends and Future Prospects*, pages 155–183. MIT Press, 1989.

[11] Hugo Liu and H. Lieberman. Toward a programmatic semantics of natural language. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 281–282, Sept 2004.

[12] Hugo Liu and Henry Lieberman. Programmatic semantics for natural language interfaces. pages 1597 – 1600, Portland, OR, United states, 2005. Brainstorming;English sentences;Linguistic structure;Natural language interfaces;Natural language queries;Natural languages;Program code;Programmatic semantics;Programming problem;Recursive structure;Storytelling;Subject matters;User study;.

[13] Henry Lieberman and Hugo Liu. *Feasibility Studies for Programming in Natural Language*, pages 459–473. Springer Netherlands, Dordrecht, 2006.

[14] J. F. Pane and B. A. Myers. Usability issues in the design of novice programming systems. pages 84p –, 1996/08/.

[15] Rada Mihalcea, Hugo Liu, and Henry Lieberman. Nlp (natural language processing) for nlp (natural language programming). In Alexander Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, pages 319–330, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[16] Dilipkumar A. Borikar Tanvi Hardeniya. Dictionary based approach to sentiment analysis - a review. *International Journal of Advanced Engineering, Management and Science*, 2, 2016.

[17] python. `https://www.python.org/about/`. Accessed: 2018-05-16.

[18] Z. Dobesova. Programming language python for data processing. In *2011 International Conference on Electrical and Control Engineering*, pages 4866–4869, Sept 2011.

[19] R. Y. Lyu, Y. H. Kuo, and C. N. Liu. Machine translation of english identifiers in python programs into traditional chinese. In *2016 International Computer Symposium (ICS)*, pages 622–625, Dec 2016.

[20] Y. C. Huei. Benefits and introduction to python programming for freshmore students using inexpensive robots. In *2014 IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE)*, pages 12–17, Dec 2014.

[21] Aleksandr Drozd, Anna Gladkova, and Satoshi Matsuoka. Python, performance, and natural language processing. In *Proceedings of the 5th Workshop on Python for High-Performance and Scientific Computing*, PyHPC '15, pages 1:1–1:10, New York, NY, USA, 2015. ACM.

[22] nltk. `https://www.nltk.org/`. Accessed: 2018-05-16.

[23] M. Lobur, A. Romanyuk, and M. Romanyshyn. Using nltk for educational and scientific purposes. In *2011 11th International Conference The Experience of Designing and Application of CAD Systems in Microelectronics (CADSM)*, pages 426–428, Feb 2011.

[24] About ros. `http://www.ros.org/about-ros/`. Accessed: 2018-05-16.

[25] P. Bouchier. Embedded ros [ros topics]. *IEEE Robotics Automation Magazine*, 20(2):17–19, June 2013.

[26] History. `http://www.ros.org/history/`. Accessed: 2018-05-16.

[27] J. Boren and S. Cousins. Exponential growth of ros [ros topics]. *IEEE Robotics Automation Magazine*, 18(1):19–20, March 2011.

[28] Morgan Quigley, Ken Conley, Brian P Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. 3, 01 2009.

[29] J. Jackson. Microsoft robotics studio: A technical introduction. *IEEE Robotics Automation Magazine*, 14(4):82–87, Dec 2007.

[30] Olivier Michel. Cyberbotics ltd. webots: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):5, 2004.

[31] ros_control ros wiki. `http://wiki.ros.org/ros_control`. Accessed: 2018-05-16.

[32] Adolfo Rodrguez Tsouroukdissian. Ros control, an overview. =https://roscon.ros.org/2014/wp-content/uploads/2014/07/ros_control_an_overview.pdf, Sep 2014. Accessed: 2018-05-16.

[33] gazebo and ros_control. `http://gazebosim.org/tutorials/?tut=ros_control`. Accessed: 2018-05-16.

[34] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. Realtime computer vision with opencv. *Queue*, 10(4):40:40–40:56, April 2012.

Appendix A

Key-terms Charts

Category	Name	Values
Subject	Car Subject	rover, car, vehicle
Subject	Camera Subject	camera
Subject	Obstacle Subject	wall, obstacle, barrier
Directions	Forward Directions	onward, ahead, forward, forwards
Directions	Backward Directions	backward, backwards, back, reverse
Directions	Reverse Directions	around
Directions	Left Directions	left, leftward
Directions	Right Directions	right, rightward
Directions	Turning Directions	Left Directions — Right Directions — Turning Directions
Directions	Pan Directions	left, right, up, down, center
Actions	Turning Actions	turn, pivot, transition, pan, rotate
Actions	Movement Actions	move, moving, go, goes, going, drive, driving, proceed, accelerate, accelerating, head

Actions	Stopping Actions	stop, halt
Units	Time Units	second, seconds
Units	Distance Units	inches, units, inch
Units	Rotation Units	degrees, degree
Magnitudes	Larger Magnitudes	greater, more, above
Magnitudes	Smaller Magnitudes	less, fewer, below
Magnitudes	Equivalent Magnitudes	same, equal
Conditionals	Inverted Conditionals	until, once
Conditionals	Normal Conditionals	while
Conditionals	Wait Conditionals	when, after
Macros	Avoid Action	avoid
Macros	Capability Action	can, able
Macros	Existence Action	is, exists, exist
Macros	Collision Action	hit, see, sees
Negations	Negations	not, isn't, no, haven't

Table A.1: Key-terms by category