Universitat de les Illes Balears

escola politècnica superior

Treball Final de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Implementation of a Programming Interface for a Mitsubishi RV-6S Industrial Robot

GABRIEL LLOBERA SALAS

Tutor
JOSE GUERRERO SASTRE

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, july 2016

CONTENTS

Acronyms

ROS	Robot Operating System
GUI	Graphical User Interface
MB4	Melfa Basic IV
API	Application Programming Interface
HTML	Hyper Text Markup Language
CSS	Cascading Style Sheets
UI	User Interface

Abstract

Mitsubishi RV-6S is the industrial robot used at the University of the Balearic Islands to teach Robotics to Industrial and Computer Engineering students. The program used to interact with the robot is COSIROP, which is the official programming software for Mitsubishi industrial robots.

COSIROP has many features that let you interact with the robot in various ways and is designed to be used in an industrial setting. However, this also means that for students it is complex to use as their first contact with the world of robotics since it contains many features that they don't need and it has been designed for robustness over usability. Furthermore, the version currently used in class only runs on Windows 2000, forcing us to use an operating system which is no longer supported and upgrading to a new version would mean an expensive investment.

Here we propose an open source implementation of the backend communications with the robot controller which runs on Ubuntu and Robot Operating System (ROS), as well as a web interface for communicating with the backend and controlling the robot. This interface is not full featured like COSIROP, but it aims to meet the needs for teaching Robotics with less complexity and an interface designed for better usability.

Keywords: ROS, Mitsubishi RV-6S, Melfa Basic IV, Robotics

RESUMEN

En la Universidad de las Islas Baleares se usa el brazo robot Mitsubishi RV-6S para impartir robótica a estudiantes de Ingeniería Industrial e Informática. Se utiliza un programa llamado COSIROP, que es el programa oficial para brazos robot de Mitsubishi.

COSIROP está hecho para ser usado en un contexto industrial y tiene muchas funcionalidades que permiten interactuar con el robot de diversas maneras. Sin embargo, esto significa que es complejo de usar como primera toma de contacto con el mundo de la robótica dado que contiene muchas funcionalidades que no se necesitan para las asignaturas y ha sido construido para ser robusto más que para ser usable. Además, la versión que se utiliza en clase solamente funciona sobre Windows 2000, lo que nos obliga a usar un sistema operativo que ya no tiene soporte oficial. Actualizar a una nueva versión del program supondría una inversión costosa.

Aquí se propone una implementación de código abierto del "backend" que funciona sobre Linux y ROS para las comunicaciones con la controladora robot, así como una interfaz web para interactuar ella y controlar el robot. Dicha interfaz no es tan completa como la del COSIROP, pero tiene como objetivo satisfacer las necesidades para la asignatura de Robótica con menos complejidad y una interfaz construida para una mejor usabilidad.

Keywords: ROS, Mitsubishi RV-6S, Melfa Basic IV, Robótica

CHAPTER 1

Introduction

1.1 Context

Industrial robots are a kind of robot system used for manufacturing and to automate jobs such as painting and welding. They are well suited to industrial settings due to their ability to move large loads, their movement accuracy and repeatability, as well as their robustness to handle long working hours without breaking down.

The University of the Balearic Islands owns a Mitsubishi RV-6S, which is a 6-axis robot arm with a repeatability of +-0.02mm, a maximum speed of 9300 mm/s, a weight of 58 kg and reach of 696mm. It is used during lab classes for the subject 22424 Robotics. This subject is compulsory for Industrial Engineering students and optional for Computer Engineering students, and it aims to introduce students to industrial robots, how they work and how they are programmed.

COSIROP is the official software to interact with Mitsubishi industrial robots. Among other things it includes a text editor for programming, a point editor to specify a coordinate and tool pose, and a jog operation to manually control the robot. The robot is programmed in MELFA Basic IV, a variant of BASIC which follows the same control flow and includes custom commands to move and control the robot.

1.2 Motivation

COSIROP is a very robust and mature software meant to be used in industrial environments, with many features that let the user control and program any Mitsubishi industrial robot. However, this also means that it is a complex program for the purpose of teaching where we only use a subset of its features. Students often forget which button they had to press or which menu contains the option that they are looking for.

Furthermore, the version of the software currently owned by the university is a version that only works on Windows 2000, which has not received updates since 2005 when official support ended. Using an outdated operating system poses a security risk, but updating to a newer version of windows would mean not only purchasing licenses for the new operating system, but it would also force the university to purchase licenses for the new version of COSIROP.

Another disadvantage of COSIROP is that it is proprietary and it does not share the implementation of the communication layer, so the low level details of the communication between the program and the robot are not public knowledge. Because it is not open source it cannot be modified to suit our needs, and because the communication layer is not documented it is not trivial to communicate with the robot outside of COSIROP.

Finally, since the computer which has COSIROP installed needs to be connected to the controller, the user cannot program the robot from anywhere else. He needs to be physically next to the robot and on the same computer.

1.3 Objectives

The objective of this project is to develop an open source program that can communicate with the robot by sending instructions that the robot can understand over serial port. This program will need to emulate the commands sent by COSIROP in order for the robot to understand them.

This program will be a generic backend that any frontend can communicate with by using ROS messages (which we will explain in section 4.2.2). The instructions sent over ROS messages should specify the task that we want the backend to perform, and they should be higher level than the ones the backend sends to COSIROP.

We will develop a simple desktop frontend that can communicate a few simple operations to the backend and can be used for testing purposes.

Since ROS messages can be sent over HTTP, we will develop a more powerful and full featured web application as the primary frontend to be used. This will enable students to control the robot from any computer in the same network, such as the computers in the lab or the student's laptop.

1.4 Document Structure

The rest of this document is structured as follows:

- In Chapter 2 we explain the current situation of the robot and software used in our lab.
- In Chapter 3 we list the project requirements that must be met.
- In Chapter 4 we discuss the communication protocol used to communicate with the controller and the implementation of our backend.
- In Chapter 5 we explain the desktop frontend that can communicate with the backend discussed in Chapter 4.
- The Chapter 6 we explain and show the web frontend and all of its functionalities.
- In Chapter 7 we show some tests that were conducted to verify that the project works as expected.

- In Chapter 8 we explain our conclusions from this project and outline future work.

CHAPTER 2

Current Framework

In this chapter we introduce the current framework and environment that is being used at UIB to teach Robotics [1]. This environment is composed of a Mitsubishi industrial robot model RV-6S, a CR2B-574 controller and COSIROP as the current software.

2.1 Mitsubishi RV-6S

The robot used during the labs is a Mitsubishi RV-6S, which can be seen in Figure 2.1. It is a 6-axis industrial robot manufactured by Mitsubishi Electric Corporation with a repeatability of +-0.02mm, a maximum speed of 9300 mm/s, a weight of 58 kg and reach of 696mm (see Table 2.1).

This robot can work with different kinds of coordinates:

1. Tool coordinates: describe the position and orientation of the robot tool in space. They are composed of (x, y, z) coordinates and three Euler angles [2] representing the tool's roll pitch and yaw rotation.

2. Joint coordinates: describe the robot's position in terms of its joint angles.

3. Base XYZ coordinates: describe the robot's position in terms of (x, y, z) coordinates relative to the robot's base.

For a visual representation of tool coordinates and base coordinates see Figure 2.2.

Item	Specifications
Repeatability	+-0.02mm
Weight	58 kg
Reach	696mm

Table 2.1: Mitsubishi RV-6S Specifications

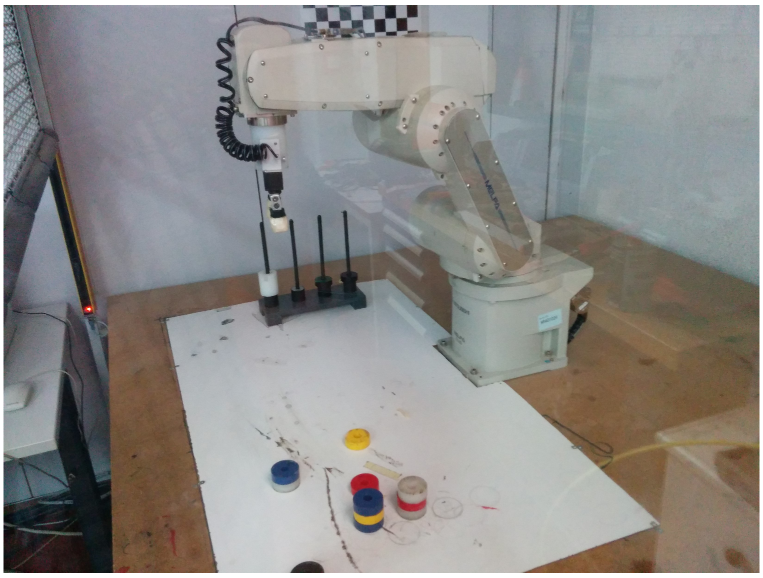


Figure 2.1: Mitsubishi RV-6S

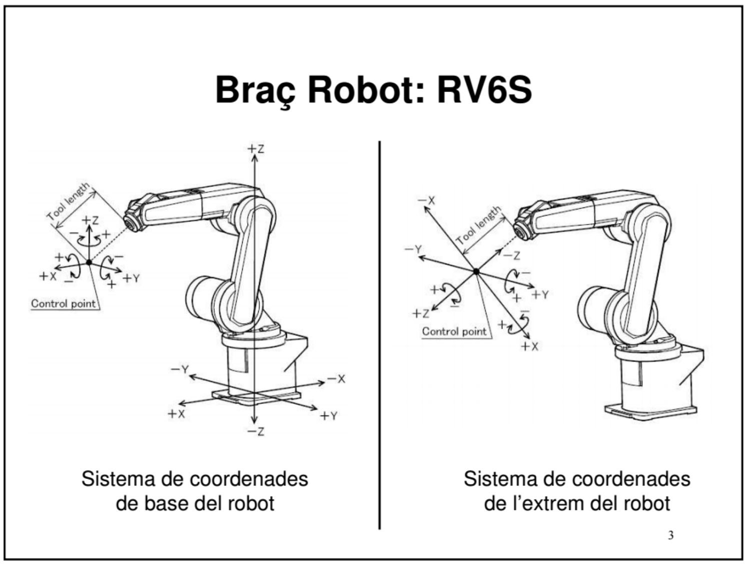

Braç Robot: RV6S
Tool length
Control point
+Z
+X
+Y
−X
−Y
−Z
Sistema de coordenades de base del robot
Sistema de coordenades de l'extrem del robot
3

Figure 2.2: Visual Representation of Base XYZ Coordinates (left) and Tool Coordinates (right)

RS-232 is used for all communication between the robot and controller. In Section 4.1 we explain the communication parameters used by our program.

2.2 CR2B-574 Controller

The controller used is a CR2B-574, one of the two controllers compatible with the RV-6S also manufactured by Mitsubishi (see Figure 4.1). The controller acts as an interface that enables communication between the computer and the robot over serial port (see Chapter 4 for more details). Its main function is to store programs and points in memory. These places in memory where programs can be stored are called slots. It is also responsible for the execution of those programs.

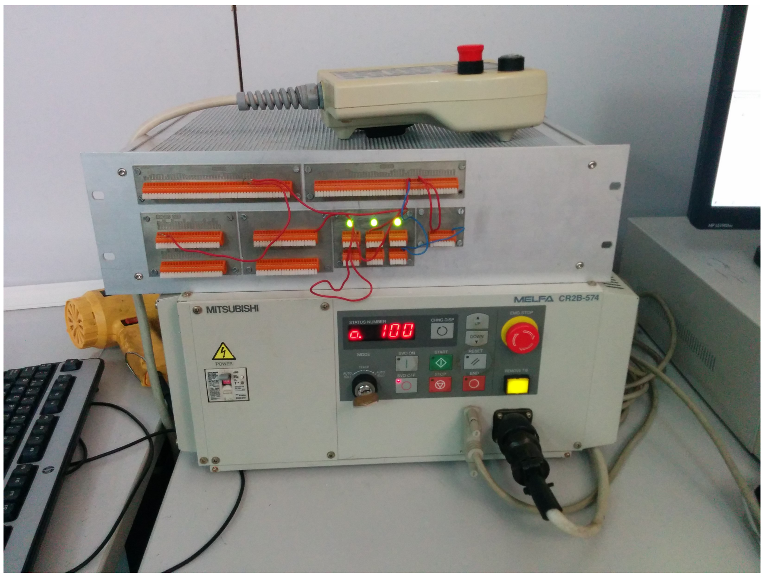


Figure 2.3: CR2B-574 Controller

The controller can operate in three modes which are: a teach mode and two auto modes. The teach mode enables the user to control the robot directly using the teaching pendant. We will also use the auto external mode which enables communication with an external program. To switch modes we need to turn the controller key (see Figure 2.4).

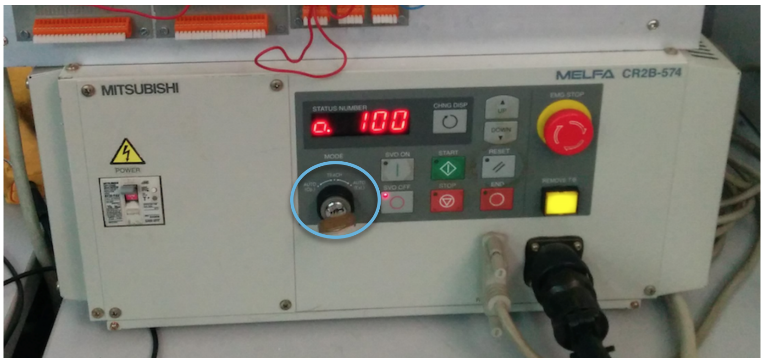
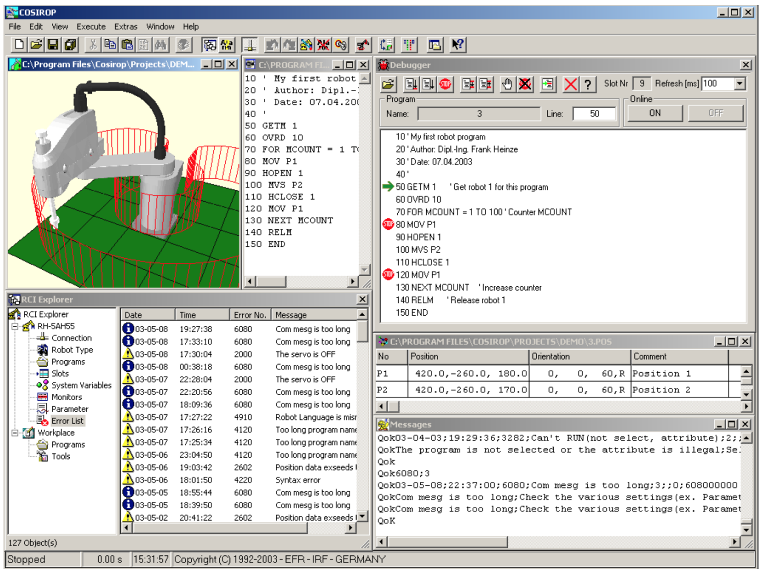


Figure 2.4: CR2B-574 Controller

2.3 COSIROP

The program used to communicate with the controller is called COSIROP, which is the official software for Mitsubishi industrial robots. In Figure 2.5 we can see that it includes a text editor (located at the top-right half) to program the robot, a point editor to specify a coordinate and tool pose (middle-right) and a simulator to test your code before sending it to the robot (top-left). Among other things it also includes a jog operation to manually control the robot.



```
10 ' My first robot
20 ' Author: Dipl.-
30 ' Date: 07.04.20(
40 '
50 GETM 1
60 OVRD 10
70 FOR MCOUNT = 1 T(
80 MOV P1
90 HOPEN 1
100 MVS P2
110 HCLOSE 1
120 MOV P1
130 NEXT MCOUNT
140 RELM
150 END
```

Program Name: 3 Line: 50

```
10 ' My first robot program
20 ' Author: Dipl.-Ing. Frank Heinze
30 ' Date: 07.04.2003
40 '
50 GETM 1      ' Get robot 1 for this program
60 OVRD 10
70 FOR MCOUNT = 1 TO 100 ' Counter MCOUNT
80 MOV P1
90 HOPEN 1
100 MVS P2
110 HCLOSE 1
120 MOV P1
130 NEXT MCOUNT   ' Increase counter
140 RELM      ' Release robot 1
150 END
```

Date	Time	Error No.	Message
03-05-08	19:27:38	6080	Com mesg is too long
03-05-08	17:33:10	6080	Com mesg is too long
03-05-08	17:30:04	2000	The servo is OFF
03-05-08	00:38:18	6080	Com mesg is too long
03-05-07	22:28:04	2000	The servo is OFF
03-05-07	22:20:56	6080	Com mesg is too long
03-05-07	18:09:36	6080	Com mesg is too long
03-05-07	17:27:22	4910	Robot Language is misr
03-05-07	17:26:16	4120	Too long program name
03-05-07	17:25:34	4120	Too long program name
03-05-06	23:04:50	4120	Too long program name
03-05-06	19:03:42	2602	Position data exseeds
03-05-06	18:01:50	4220	Syntax error
03-05-05	18:55:44	6080	Com mesg is too long
03-05-05	18:39:50	6080	Com mesg is too long
03-05-02	20:41:22	2602	Position data exseeds

C:\PROGRAM FILES\COSIROP\PROJECTS\DEMO\3.POS

No	Position	Orientation	Comment
P1	420.0,-260.0, 180.0	0, 0, 60,R	Position 1
P2	420.0,-260.0, 170.0	0, 0, 60,R	Position 2

Messages

```
Qok03-04-03;19:29:36;3282;Can't RUN(not select, attribute);2;;
QokThe program is not selected or the attribute is illegal;Se
Qok
Qok6080;3
Qok03-05-08;22:37:00;6080;Com mesg is too long;3;;0;608000000
QokCom mesg is too long;Check the various settings(ex. Paramet
QokCom mesg is too long;Check the various settings(ex. Paramet
QoK
```

Copyright (C) 1992-2003 - EFR - IRF - GERMANY

Figure 2.5: COSIROP

2.4 Melfa Basic IV

The robot is programmed in MELFA Basic IV, a variant of BASIC which follows the same control flow and includes custom commands to move and control the robot. For a sample Melfa Basic IV program see the listing below. This program starts the servo (SERVO ON), sets the speed (SPD 20) and joint override (OVRD 20), moves to two points (MOV Pa and MOV Pb) which should have previously been loaded with COSIROP and opens the actuator (HOPEN 1). Note that every line needs to start with a number in increasing order. See the program below:

```
10 SERVO ON
20 SPD 20
30 OVRD 20
40 MOV Pa
50 MOV Pb
60 HOPEN 1
70 END
```

For the previous program to work it is necessary to also define points Pa and Pb in a separate file. An example points file has been provided below:

```
DEF POS Pa=(584.73,141.04,385.25,−163.74,1.11,−78.61)(7,0)
DEF POS Pb=(225.48,−363.55,683.37,−128.32,2.12,−123.50)(7,0)
```

Those points in the previous listing are defined as a tool pose, which describes the position of the robot's tool in space. The first three numbers represent the (x, y, z) coordinates and the other three numbers represent the tool's roll pitch and yaw.

CHAPTER 3

PROJECT REQUIREMENTS

To make the project more modular and easier to develop we broke it down into three separate parts. We now present the requirements individually for each part. A diagram showing the overall project architecture and interactions between components is shown in Figure 3.1. We can see that the system is composed of two main parts: the backend and the frontend. The backend runs on ROS and communicates with the controller, while the frontend communicates with the backend and is independent of the implementation. To demonstrate this we create two separate frontends: an RQt application and a web application.

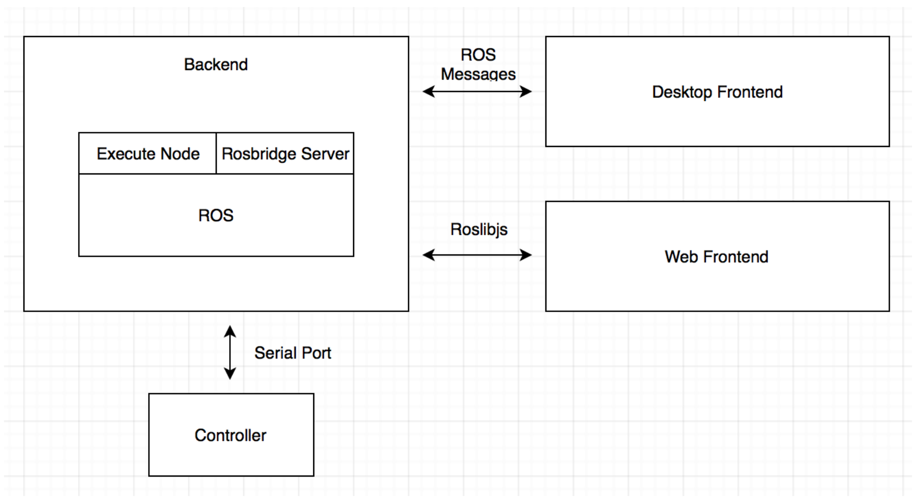


Figure 3.1: Project Architecture

3.1 Backend

The backend will be responsible for all communication with the robot controller. It will receive messages from the frontend and send commands to the controller over serial port (for more information see Section 4.3).

1. The system must load a program onto the first slot in the controller correctly.
2. The system must load points onto the first slot in the controller correctly.
3. The system must delete the program and points in the first slot of the controller.
4. The system must execute an arbitrary command on the controller.
5. The system must be able to perform jog operations with coordinates relative to the tool.
6. The system must be able to run the program loaded on Slot 1.
7. The system must be able to request the current joint state and broadcast it on a ROS channel (see Section 4.2).
8. The system must be able to request the current tool pose and broadcast it on a ROS channel.

3.2 Desktop Frontend

The desktop frontend will be an application that runs on Ubuntu and ROS. It will provide a minimal interface to interact with the backend.

1. The program must be able to send a Melfa Basic IV program to the backend for loading.
2. The program must be able to send a points file to the backend for loading.
3. The program must show an error if the user is trying to send a program file with the wrong extension.
4. The program must show an error if the user is trying to send a points file with the wrong extension.
5. The program must be able to instruct the backend to run the currently loaded program.

3.3 Web Frontend

The web frontend provides a more complete and portable way to interact with the backend and requires no installation (see Chapter 6).

1. The web interface must display the connection status.

2. If the connection status is disconnected the user must be able to try to reconnect without refreshing the page.
3. The web interface must contain a text editor where Melfa Basic IV code must be entered.
4. The text editor must be able to perform syntax checking on the fly.
5. The text editor must perform static analysis to warn of the uses of undeclared points.
6. The text editor must perform static analysis to warn of the uses of undeclared labels.
7. The web interface must permit disabling on the fly syntax checking.
8. The web interface must permit manually clearing all syntax errors.
9. The web interface must permit manual syntax checking.
10. The web interface must be able to send a Melfa Basic IV program to the backend for loading.
11. The web interface must be able to load a Melfa Basic IV file from the computer.
12. The web interface must assist in entering point positions.
13. The web interface must be able to display the current robot position.
14. The web interface must be able to instruct the backend to move the robot to a defined point.
15. The web interface must be able to send a points file to the backend for loading.
16. The web interface must be able to load a points file from the computer.
17. The web interface must be able to instruct the backend to run the currently loaded program.
18. The web interface must be able to delete the currently loaded program and points.
19. The web interface must include a jog operation to move the robot tool.
20. The jog operation must permit opening and closing the actuator.

CHAPTER 4

COMMUNICATION PROTOCOL

To enable communication with the robot, the computer running COSIROP needs to be connected to a controller (see Figure 4.1), as pointed out in Section 2.2. The task of this controller is to receive commands from the computer and act as an interface between the computer and the robot. This means that COSIROP does not communicate directly with the robot, but rather it sends commands over serial port to the controller where they will be interpreted. The controller will then instruct the robot to take the appropriate action. These actions can include turning the servo on or off, executing a program or controlling the arm manually with the jog operation using a teaching pendant (see Figure 4.2) and so on. Our Mitsubishi RV-6S is controlled by the CR2B-574 controller box, also manufactured by Mitsubishi Electric Corporation.

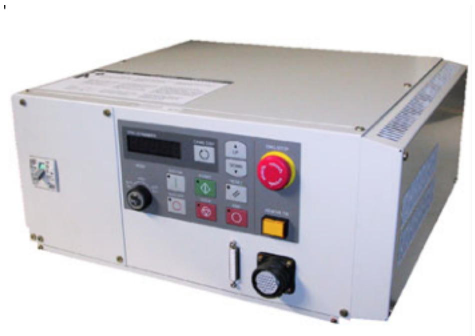


Figure 4.1: CR2B-574 Controller

4.1 RS-232 Communication Settings

We have mentioned earlier that all communication between the computer and the controller happens over serial port using RS-232. Here we will talk more specifically about the communication parameters that are required to achieve successful communication.

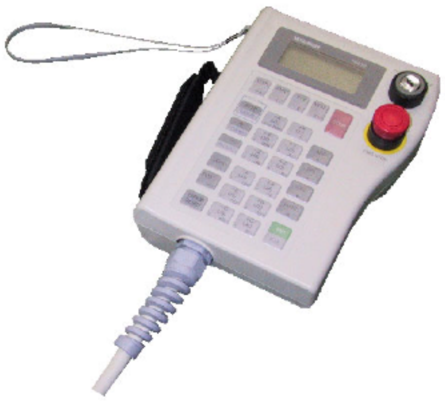


Figure 4.2: Teaching Pendant

Parameter	Value
Bits per second (bps)	9600 baud
Data bits	8 bits
Parity	Even

Table 4.1: RS-232 parameters

In order to communicate successfully with the controller we need to set the parameters of the serial port communication correctly. See Table 4.1 for the parameters we have obtained using COSIROP.

4.2 Robot Operating System

To simplify the communication process between different components of the same system that need to interact with each other the ROS modules have been created. They provide an interface for message passing and inter-process communication to facilitate interaction.

4.2.1 History and introduction

ROS [3] is an open source software framework designed to simplify writing robot software and enable collaboration. It was originally developed in 2007 at the Stanford Artificial Intelligence Laboratory and it later continued to be developed by Willow Garage.

By using the collection of tools and libraries offered by ROS, programs can be built on top of other people's work and avoid having to reinvent the wheel every time. By relying on packages written by other contributors, the user can focus on working on his area of expertise and writing a good implementation which other ROS users will be able to use and build upon in the future.

4.2.2 Communications Infrastructure: Messages

One of the core components of ROS is the communications infrastructure. To facilitate modularity ROS works as a distributed system where each component is independent and communicates with others by message passing. This means that at any point in time the user would be able to substitute one component for another as long as it can understand and produce the same messages as its predecessor. Some advantages of this approach are that components are independent of each other, easily interchanged and they don't rely on implementation details of other components, which leads to smaller components which are more easily developed, verified and maintained.

The communication model used by ROS is the publish-subscribe pattern (see Figure 4.3), where messages are not sent to specific receivers, but rather published to channels.

When a node wishes to communicate something it will publish a ROS message on a particular channel. All other nodes which are subscribed to this channel will be notified and they will be able to receive the message and process it appropriately. As a result of this, publishers are decoupled from subscribers and they don't need to be aware of each others existence, making it trivial to add or remove components. The resulting components can be developed in isolation from the others and are thus easier to reason about and test.

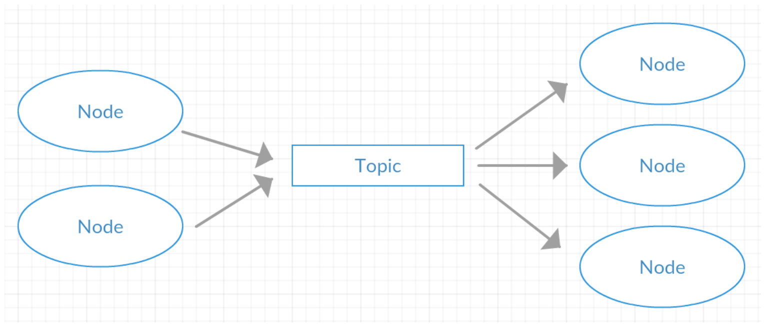


Figure 4.3: Publish subscribe pattern

4.2.3 Nodes and services

ROS nodes are processes that perform computations and communicate with each other through ROS messages using the publish/subscribe model. They are combined together into a graph and can be uniquely identified by a graph resource name. For a visual representation of the above-mentioned graph see Figure 4.4). They help to create distributed systems which are easier to develop, maintain and scale.

ROS services are designed for request response interactions, for which the publish/-subscribe model is not adequate. Requests and replies are ROS messages. A service is provided under a string name, when the client calls it it performs a specific function and the client waits for the response.

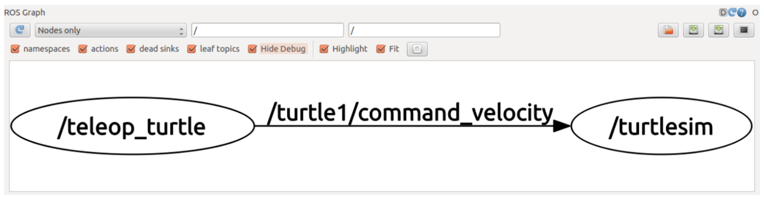


Figure 4.4: Nodes Graph

4.3 Command Protocol

COSIROP communicates with Mitsubishi RV-6S by sending commands using a proprietary protocol which has not been provided by the manufacturer. In order to achieve communication with the robot from an external program it was necessary to reverse engineer the protocol and reproduce it in out program. We did not start from scratch as a first approach had been developed for ROS in C++ by Stwirth et al. [4]. This code could successfully communicate with the robot arm to execute a single arbitrary instruction by sending commands over RS-232 with the expected protocol.

However, this project did not fully fit our needs since we needed to upload and execute full programs and perform other operations which weren't defined, not just execute single instructions. We forked the project and continued development to adapt it to our needs without having to start from scratch. Besides Stwirth's code other nodes were used such as Rosbridge

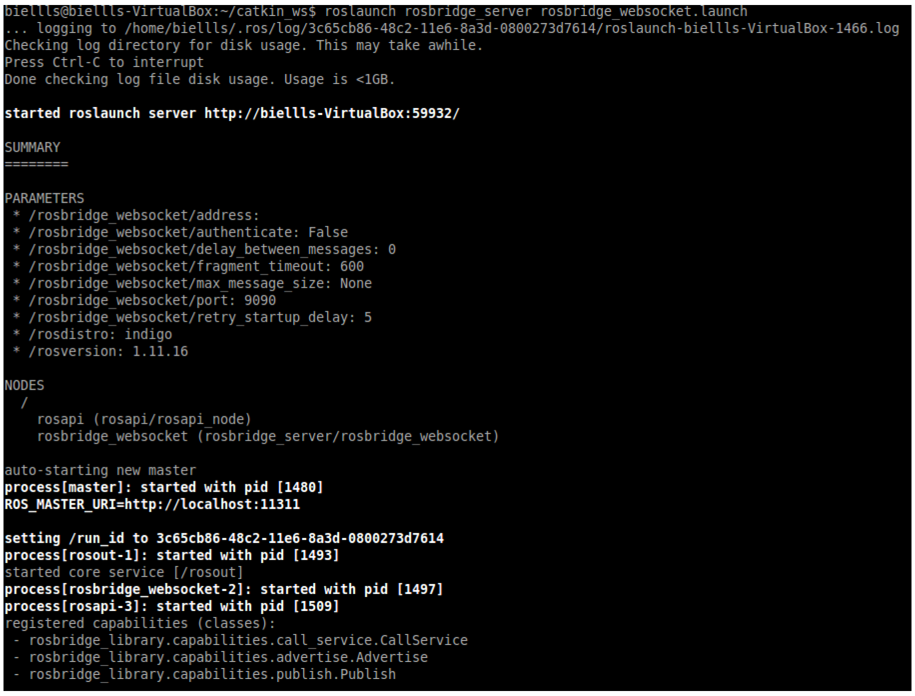
A description of the developed nodes, together with a more detailed description of the communication protocol will be given in this section.

4.3.1 Rosbridge Suite

The Rosbridge suite provides a JSON Application Programming Interface (API) that enables external programs that are not running under ROS to send ROS messages. In our case we will use the Rosbridge Server, which is a modified HTTP server that runs under ROS, to receive messages from our web client. Our web frontend will use a javascript library called Roslibjs to communicate with Rosbridge Server. This enables us to send and receive ROS messages from our frontend without having to physically run on the same machine as the backend. We can see an instance of Rosbridge Server running in Figure 4.5.

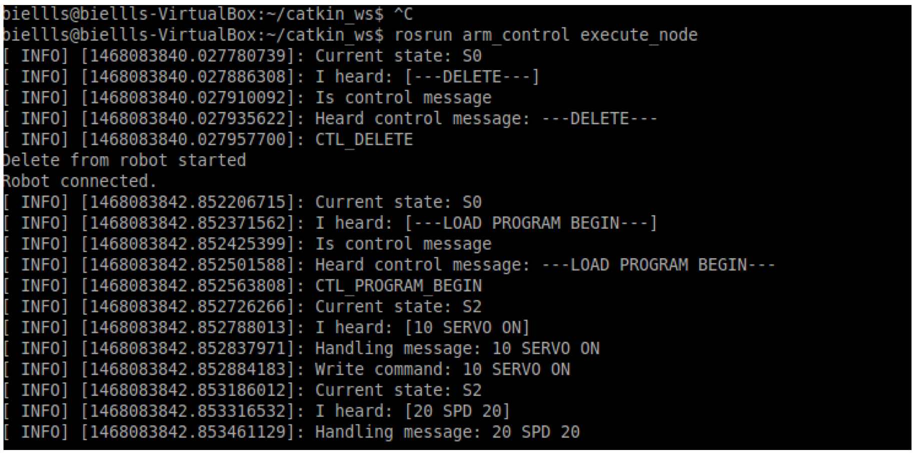
4.3.2 Execute Node

In order to replace COSIROP we will need to implement additional functionality in the Arm Control project. We will extend the project by creating a ROS node called Execute Node which will serve as the backend for all our commands. Its task will be to receive high level commands in the form of ROS messages and execute them by sending a series of lower level commands that the robot can understand over serial port. This node will act as a server by listening for requests and handling them appropriately, and it's completely independent of the program that sends the requests. We can see an instance of Execute Node receiving a program from the frontend in Figure 4.6. In Figure 4.7 Execute Node is sending a points file to the controller.



```
biellls@biellls-VirtualBox:~/catkin_ws$ roslaunch rosbridge_server rosbridge_websocket.launch
... logging to /home/biellls/.ros/log/3c65cb86-48c2-11e6-8a3d-0800273d7614/roslaunch-biellls-VirtualBox-1466.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://biellls-VirtualBox:59932/

SUMMARY
========

PARAMETERS
 * /rosbridge_websocket/address:
 * /rosbridge_websocket/authenticate: False
 * /rosbridge_websocket/delay_between_messages: 0
 * /rosbridge_websocket/fragment_timeout: 600
 * /rosbridge_websocket/max_message_size: None
 * /rosbridge_websocket/port: 9090
 * /rosbridge_websocket/retry_startup_delay: 5
 * /rosdistro: indigo
 * /rosversion: 1.11.16

NODES
  /
    rosapi (rosapi/rosapi_node)
    rosbridge_websocket (rosbridge_server/rosbridge_websocket)

auto-starting new master
process[master]: started with pid [1480]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 3c65cb86-48c2-11e6-8a3d-0800273d7614
process[rosout-1]: started with pid [1493]
started core service [/rosout]
process[rosbridge_websocket-2]: started with pid [1497]
process[rosapi-3]: started with pid [1509]
registered capabilities (classes):
 - rosbridge_library.capabilities.call_service.CallService
 - rosbridge_library.capabilities.advertise.Advertise
 - rosbridge_library.capabilities.publish.Publish
```

Figure 4.5: Rosbridge Server



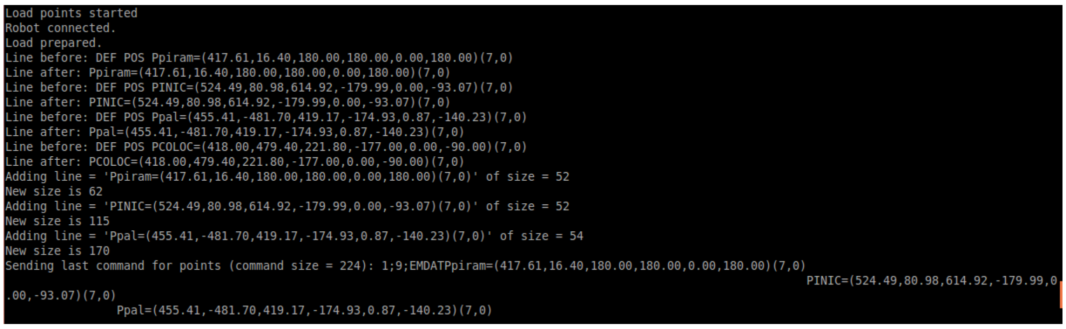
```
biellls@biellls-VirtualBox:~/catkin_ws$ ^C
biellls@biellls-VirtualBox:~/catkin_ws$ rosrun arm_control execute_node
[ INFO] [1468083840.027780739]: Current state: S0
[ INFO] [1468083840.027886308]: I heard: [---DELETE---]
[ INFO] [1468083840.027910092]: Is control message
[ INFO] [1468083840.027935622]: Heard control message: ---DELETE---
[ INFO] [1468083840.027957700]: CTL_DELETE
Delete from robot started
Robot connected.
[ INFO] [1468083842.852206715]: Current state: S0
[ INFO] [1468083842.852371562]: I heard: [---LOAD PROGRAM BEGIN---]
[ INFO] [1468083842.852425399]: Is control message
[ INFO] [1468083842.852501588]: Heard control message: ---LOAD PROGRAM BEGIN---
[ INFO] [1468083842.852563808]: CTL_PROGRAM_BEGIN
[ INFO] [1468083842.852726266]: Current state: S2
[ INFO] [1468083842.852788013]: I heard: [10 SERVO ON]
[ INFO] [1468083842.852837971]: Handling message: 10 SERVO ON
[ INFO] [1468083842.852884183]: Write command: 10 SERVO ON
[ INFO] [1468083842.853186012]: Current state: S2
[ INFO] [1468083842.853316532]: I heard: [20 SPD 20]
[ INFO] [1468083842.853461129]: Handling message: 20 SPD 20
```

Figure 4.6: Execute Node Running

Since the message protocol is proprietary we had to use reverse engineering by hooking up a listener on the serial port used by COSIROP and then reproducing the messages from our code. Every message starts with 1;1 or 1;9 followed by the name of the instruction to execute, as can be seen in Figure 4.8

Before showing the operations that the node can perform, we provide in Table 4.2 an explanation of the instructions that we will send to the controller, which have been obtained by reverse engineering.

Here we show the high level operations we have defined and the corresponding translation into instructions that the robot understands:



```
Load points started
Robot connected.
Load prepared.
Line before: DEF POS Ppiram=(417.61,16.40,180.00,180.00,0.00,180.00)(7,0)
Line after: Ppiram=(417.61,16.40,180.00,180.00,0.00,180.00)(7,0)
Line before: DEF POS PINIC=(524.49,80.98,614.92,-179.99,0.00,-93.07)(7,0)
Line after: PINIC=(524.49,80.98,614.92,-179.99,0.00,-93.07)(7,0)
Line before: DEF POS Ppal=(455.41,-481.70,419.17,-174.93,0.87,-140.23)(7,0)
Line after: Ppal=(455.41,-481.70,419.17,-174.93,0.87,-140.23)(7,0)
Line before: DEF POS PCOLOC=(418.00,479.40,221.80,-177.00,0.00,-90.00)(7,0)
Line after: PCOLOC=(418.00,479.40,221.80,-177.00,0.00,-90.00)(7,0)
Adding line = 'Ppiram=(417.61,16.40,180.00,180.00,0.00,180.00)(7,0)' of size = 52
New size is 62
Adding line = 'PINIC=(524.49,80.98,614.92,-179.99,0.00,-93.07)(7,0)' of size = 52
New size is 115
Adding line = 'Ppal=(455.41,-481.70,419.17,-174.93,0.87,-140.23)(7,0)' of size = 54
New size is 170
Sending last command for points (command size = 224): 1;9;EMDATPpiram=(417.61,16.40,180.00,180.00,0.00,180.00)(7,0)
                                                                                                                    PINIC=(524.49,80.98,614.92,-179.99,0
.00,-93.07)(7,0)
                Ppal=(455.41,-481.70,419.17,-174.93,0.87,-140.23)(7,0)
```

Figure 4.7: Execute Node Sending Points

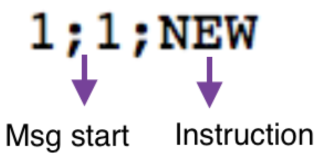


Figure 4.8: Command Structure

Command	Action
1;1;OPEN=NARCUSR	Starts communication with the robot
1;1;PARRLNG	Part of initialization routine
1;1;PDIRTOP	Part of initialization routine
1;1;PPOSF	Part of initialization routine
1;1;KEYWDptest	Part of initialization routine
1;1;NEW	Begins a load operation
1;1;LOAD=1	Tells the controller to load into Slot 1
1;1;PRTVERLISTL	Required before sending program or point data
1;1;PRTVEREMDAT	Required before sending program or point data
1;9;LISTL<	List the currently loaded program
1;9;EMDAT	The data is sent along with this command
1;1;SAVE	Save data into the previously specified slot
1;1;RUN	Run the program
1;1;CNTLON	Enable manual control
1;1;CNTLOFF	Disable manual control
1;1;FDEL1	Delete program and points from the previously specified slot

Table 4.2: Robot Commands

Init Robot for Load

Before can executing any of the other commands we need to connect with the robot. The project already provided a method to connect to the robot for manual control and execution. However, since we needed to load and execute whole programs we had to implement a new initialization function, which must be sent before every of the operations described in the following subsections. The command sequence is:

```
1;1;OPEN=NARCUSR
1;1;PARRLNG
1;1;PDIRTOP
1;1;PPOSF
1;1;PARMEXTL
1;1;KEYWDptest
```

Delete Program and Points

We will delete the currently loaded program and points from Slot 1 on the controller with the following command sequence:

```
1;1;SAVE
1;1;CNTLON
1;1;RSTPRG
1;1;CNTLOFF
1;1;FDEL1
```

Upload Program

We will upload the given program to Slot 1 on the controller. Suppose we are given the following program:

```
10 SERVO ON
20 END
```

We will send the following commands:

```
1;1;NEW
1;1;LOAD=1
1;1;PTRVERLISTL
1;1;PTRVEREMDAT
1;9;LISTL<
1;9;EMDAT10\v20
1;9;EMDAT10 SERVO ON\v20 END
1;1;SAVE
```

If the program is too long to be sent in one message it will be broken up into multiple messages which will be sent consecutively, each of them starting with 1;9;EMDAT.

Upload Points

We will upload the given points to Slot 1 on the controller. Suppose we are given the following points:

```
DEF POS P1=(584.73,141.04,385.25,-163.74,1.11,-78.61)(7,0)
DEF POS P2=(225.48,-363.55,683.37,-128.32,2.12,-123.50)(7,0)
```

We will send the following messages

Message	Meaning
—LOAD PROGRAM BEGIN—	The following messages will be lines from an
.MB4 program	
—LOAD PROGRAM BEGIN—	Stop waiting for program lines
—RUN PROGRAM—	Run the program currently loaded in memory
—MOVE JOINT STATE—	The following messages will be commands to
move the robot arm	

Table 4.3: Some Execute Node Messages

```
1;1;NEW
1;1;LOAD=1
1;1;PTRVERLISTL
1;1;PTRVEREMDAT
1;9;LISTL<
1;9;EMDATP1=(584.73,141.04,385.25,−163.74,1.11,−78.61)(7,0)\v
                                  P2=(225.48,−363.55,683.37,−128.32,2.12,−123.50)(7,0)
1;1;SAVE
```

If the points file is too long to be sent in one message it will be broken up into multiple messages which will be sent consecutively, each of them starting with 1;9;EMDAT.

4.3.3 Execute Node Messages

We have mentioned that Execute Node receives higher level messages and translates them into serial port messages that it can send to the controller. In this section we will explain how those messages are structured and give a few examples.

Message Structure

We define a control message as a message that does not contain information itself, but it provides context so that the program can interpret the next messages. Every control message has the following form: "—MY CONTROL MESSAGE—" where "MY CONTROL MESSAGE" can be one of the commands explained in Table 4.3. It starts and ends with "—", and contains a description of the instruction. In addition to control messages, information messages can also be sent. The full list of commands can be seen in Section 9.1 of the Appendix. In the following example we send a control message to tell the node that the following messages will be the lines of a program. Then we send each program line in a different message and we finally send another control message to tell the node that we have finished sending the program. This is the complete example:

```
———LOAD PROGRAM BEGIN———
10 SERVO ON
20 SPD 10
30 MOV Pa
40 END
———LOAD PROGRAM END———
```

4.3.4 State Machine

The message structure explained in the previous section forces us to maintain state to know what kind of message we should expect next. For example, if we just received a control message to begin loading a program, we don't expect to receive another control message telling us to load points. In this situation we should throw an error. Similarly, if we are in the initial state we expect to receive a control message telling us what to do, we don't expect to receive a message containing Melfa Basic IV (MB4) code.

To achieve this we decided to implement Execute Node as a state machine where each state knows what kind of messages it should expect next. This approach is simple to implement and it lets us detect errors and have some context which simplifies the interpretation of messages and removes ambiguity.

There is an initial state which expects a control message telling it what to do. This control message might correspond to an immediate command, like "—RUN PROGRAM—", or it might tell it what to expect in the next messages, such as "—LOAD PROGRAM BEGIN—". In the first case we will perform an action and stay in the initial state. In the second case we will transition to a new state where we will handle the following messages appropriately until we receive a control message telling us to return to the initial state. Figure 4.9 shows this state machine where the meaning of each state is:

```
S0: Initial State
S1: Execute Single Instruction State
S2: Load Program State
S3: Load Points State
S4: Jog Operation State
```

- Loading a program and points: When we receive a request to load a program (—LOAD PROGRAM BEGIN—) we transition to S1 where we expect to receive the program line by line in the following messages. We will write each line to a local file and when we receive a message telling us that the whole program has been sent (—LOAD PROGRAM END—), we will send the program to the controller telling it to store it in Slot 1 and promptly return to S0. Loading points follows an identical process but requires different begin and end control messages (—LOAD POINTS BEGIN— and —LOAD POINTS END—).

- Running a program or Deleting program and points: Requests to run or delete a program do not cause a change in state. They cause the node to send appropriate instructions to the controller immediately and stay in S0 waiting for further commands.

- Request for robot position: When we receive a request for the robot position we will send the command "—REQUEST TOOL POSE—" or "—REQUEST JOINT STATE—" to the controller and retrieve the robot position from the response. We will then publish this response to a ROS channel so that the client may retrieve it.

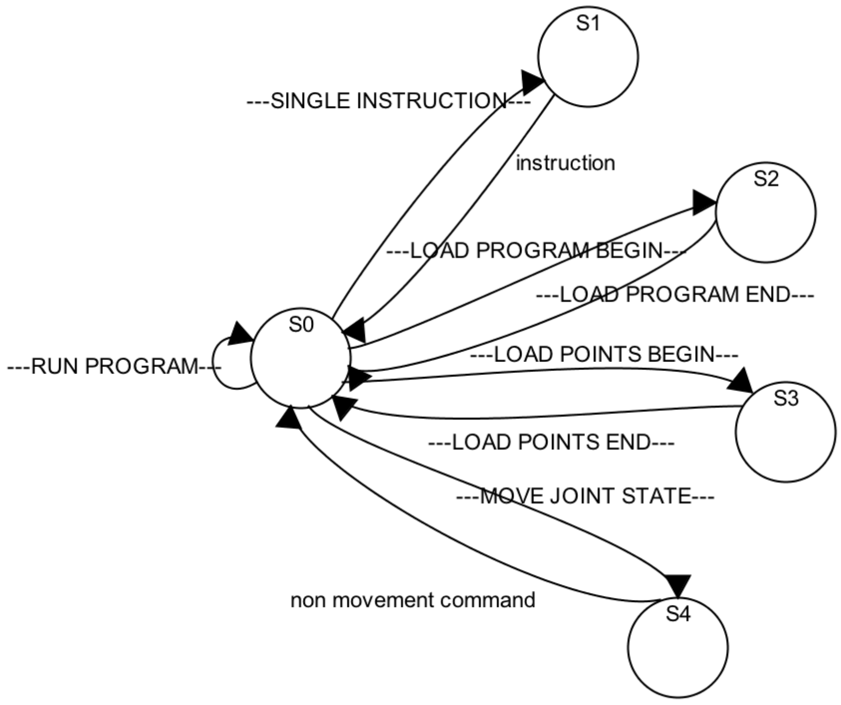


Figure 4.9: State Machine Diagram

4.3.5 Jog Operation State and Multithreading

This is possibly the most complex state. When we receive a request to move the joint we enter jog operation mode, where we expect to receive commands to move the robot tool. Such commands can tell the robot to move its tool in the x, y or z direction as well as telling it to open or close it.

Our first approach was to handle each request separately. While it worked correctly, due to the way Arm Control is implemented the robot was initialized and deinitialized after every command, which caused the jog operation to be unacceptably slow for practical real time operation.

It was obvious that the only solution to achieve a reasonable operation speed was to keep the robot initialized and waiting for requests so that it does not have to keep re-initializing after every command. Unfortunately, due to the way Arm Control was built we cannot keep the robot initialized across different C++ functions, and to do so we would need to rebuild it from the ground up. We also cannot stay within one function due to the way ROS works, which separates the message listener function from the message handler, and we cannot listen for new messages within the handler.

The only viable solution was to open a new thread of execution when we receive a command to enter the move joint state. This way we can keep receiving ROS messages in the main thread while we execute their effects on the jog operation thread. We now present pseudo-code of our solution:

```
function jog_operation()
```

```
  while jog_operation_active
    //Perform operations or wait for instructions
    sleep(0.1)
  end while
end jog_operation

function message_handler(msg)
  if move_joint_state_message(msg)
    if not jog_operation_active
      jog_operation_active = True
      t = new Thread(jog_operation)
      t.start()
    end if
  else if move_joint_command(msg)
    //Tell jog operation to perform action
  else
    if jog_operation_active
      jog_operation_active = False
      t.join()
    end if
    //Handle message
  end if
end message_handler
```

By staying within the jog operation function the whole time we can perform as many jog operations as we want and only have to initialize the robot once when we receive the first command. There is still one problem left unresolved due to the fact that the jog operation is designed to be operated manually on the client by the user.

These messages will be generated by the user from a graphical interface which will be introduced in Chapter 6. As will be seen, this interface is similar to the teaching pendant introduced in Section 2.2. If the user presses a button for a few seconds this could potentially generate hundreds of messages before the tool has finished its first movement. As a result of the accumulated messages the robot would keep moving in that direction long after the user has stopped pressing the button. This is not practical for real use, so we had to think of a way to discard extra messages and only move when the user really wants to.

In order to solve this problem we first need to define which messages should be processed and which should be ignored. We propose that a reasonable definition of a message that should be treated is a message that was generated when the tool was not already moving. Another way to think of this is that if the tool is not finished moving and we ask it to move again, this message should be discarded and not accumulated.

As a more formal definition, we will treat a message only if the timestamp of its creation is greater than the timestamp of the last finished move.

Solution: Handling Multiple Messages

For every possible jog operation we will have a global variable representing the timestamp of the last message that requested this operation. Every time we receive a message requesting a move operation we will update the corresponding global variable to the timestamp of this message. Those variables will always contain the latest timestamp that the operation was requested, so to know if we need to move the tool at a given instant we just need to check if any of the timestamps of the local variables is greater than the timestamp of the last finished move. We can see an example message in Figure 4.10.

$$---MOV\ TOOL\ \ +X---1468360644$$

Figure 4.10: Example Move Joint Message with Timestamp

We offer a more complete pseudo-code of the implementation in Section 9.2 of the Appendix.

CHAPTER 5

RQT FRONTEND

After describing the communication protocol in Chapter 4 and the implementation of a backend that can communicate with the controller, in this section we present a simple frontend in RQt (ROS Qt) that communicates with it. The purpose of this frontend is not to be highly usable or full featured, but rather to provide a fast and easy way to test changes in the backend. It also serves our purpose of illustrating how the backend and frontend are completely independent and how we can easily create new clients that communicate with our backend.

RQt is a framework for designing Graphical User Interface (GUI)s based on the open source cross-platform application framework Qt. It implements GUI tools in the form of plugins and offers good integration with ROS, which is why we chose it as our first frontend to implement.

Since it it based on the mature framework Qt we can use any of the available QT tools to create our GUI. We used Qt creator as a form builder to create the layout of the interface graphically. This layout is generated as an XML file that can be used in any Qt or RQt application. In our case we decided to program with Python to create all the application logic behind the interface.

RQT provides an API to write the application logic in either C++ or Python. This lets the user define handlers for every action and is independent of the look of the GUI, which is specified in an XML file. We decided to implement ours in Python given the simplicity of the language and the fact that the amount of code we had to write was small enough that we didn't need static type checking.

Every action executed by the user is handled by a Python function that sends the appropriate ROS messages to the backend. In Figure 5.2 we can see the handler that is invoked when the user presses the "Load program" button. This function checks that the file has the correct extension, sends an initialization control message, sends the file line by line and sends a control message indicating it has finished sending the file.

We can observe in Figure 5.3 that the interface is very simple. We can select a program and points file from the file system (by clicking the "Choose program" and "Choose points" buttons respectively) and tell the backend to load them. We can also

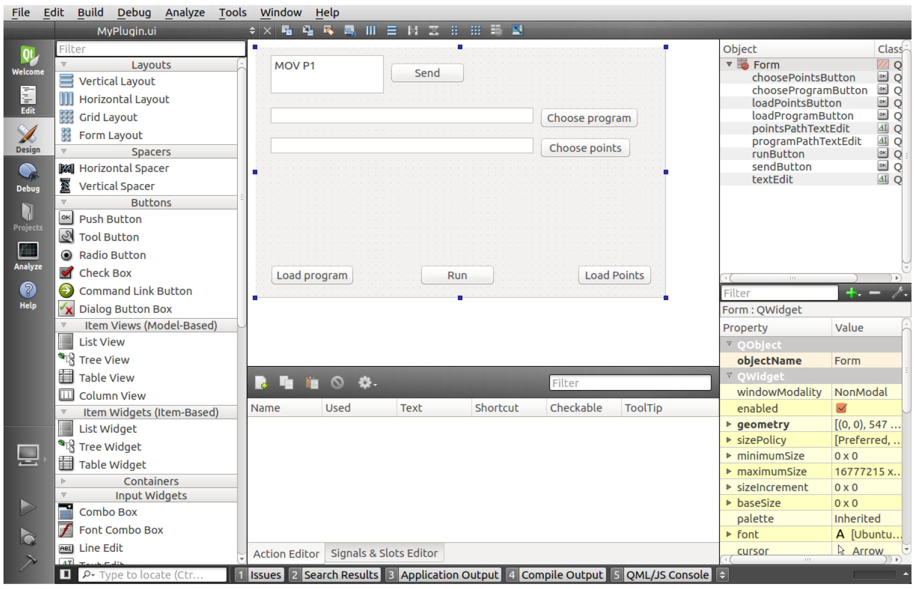


Figure 5.1: GUI Design with QT Creator

```
def _loadProgram(self):
    fname = self._widget.programPathTextEdit.toPlainText()
    if fname == "":
        sys.stderr.write("No file specified\n")
        QMessageBox.warning(self._widget, "Error loading program", "Please specify a file")
        return

    if not fname.endswith(".MB4"):
        sys.stderr.write("Please choose .MB4 file\n")
        QMessageBox.warning(self._widget, "Error loading program", "Wrong extension: Please choose .mb4 file")
        return

    self.pub.publish('---LOAD PROGRAM BEGIN---')
    self.publish_file(fname)
    self.pub.publish('---LOAD PROGRAM END---')
```

Figure 5.2: RQT Python Handler for Loading a Program

send an arbitrary MB4 command to the backend and run the currently loaded program.

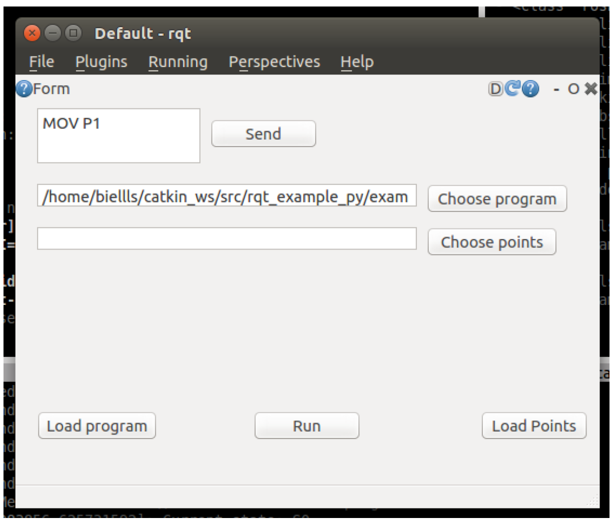
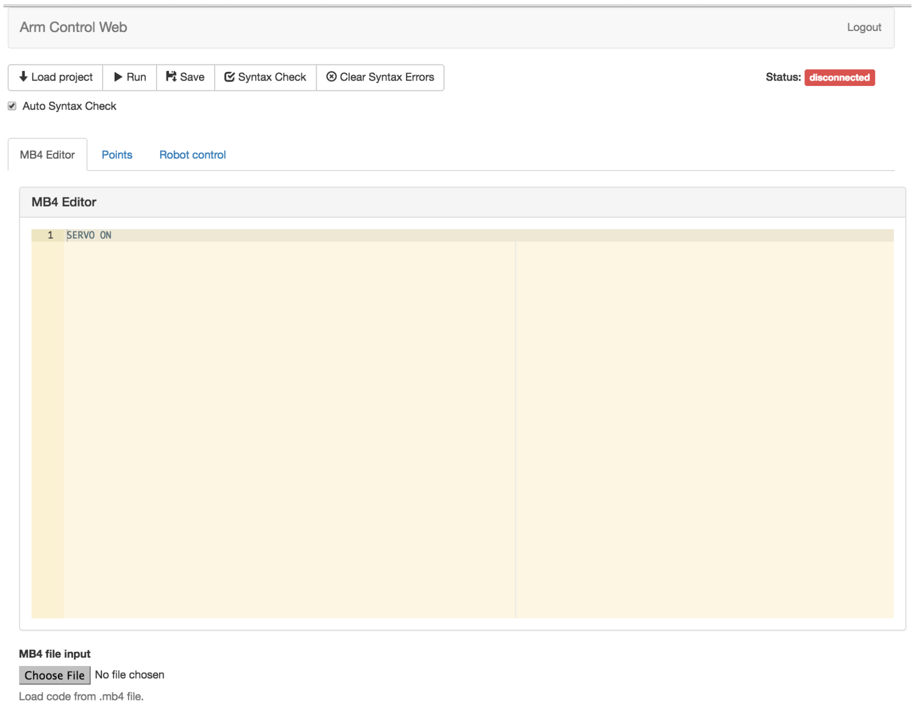

Default - rqt
File Plugins Running Perspectives Help
Form
MOV P1
Send
/home/biellls/catkin_ws/src/rqt_example_py/exam
Choose program
Choose points
Load program
Run
Load Points

Figure 5.3: RQT Frontend GUI

CHAPTER 6

WEB FRONTEND

We now have a fully operational backend that handles the details of communicating with the controller and we have tested it with a simple frontend to make sure that it works correctly. The next step is to create a more usable frontend which makes use of all the features supported by the backend. We chose to make this frontend a



```
SERVO ON
```

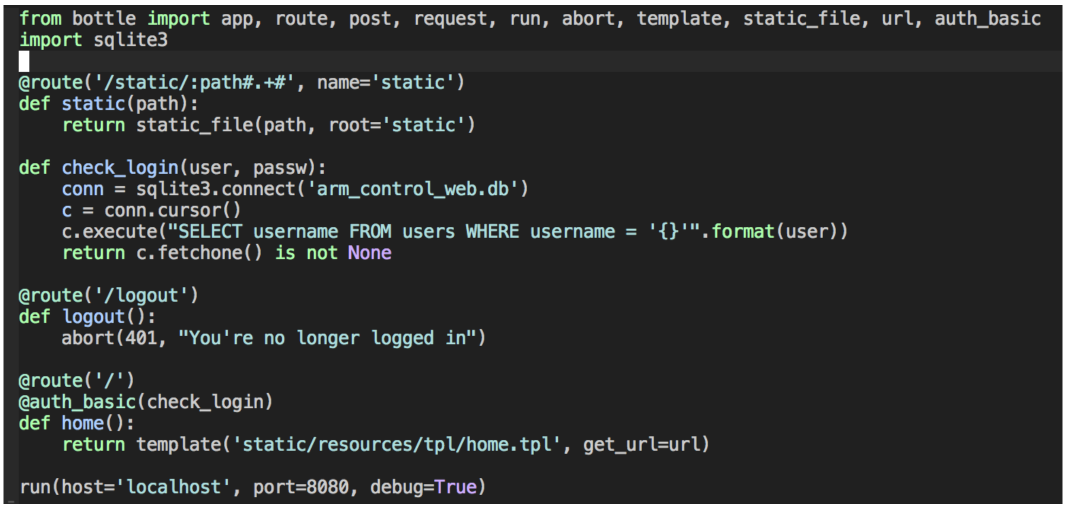
Figure 6.1: Final Web Frontend for Arm Control

web application because it would give students flexibility to control the robot from any computer they wish as long as it is connected to the same subnet. Also, web applications have reached a state of maturity where there are many tools and frameworks that simplify the development process and enable the developer to create websites with design and usability that rivals or surpasses desktop applications. Figure 6.1 shows the final User Interface (UI) of our application.

6.1 Web Server and Roslibjs

Since web applications contain almost all the program logic on the client side, the web server will be very simple. It will mostly just validate user access and serve content, without having to keep any other session state.

We decided to use the Bottle Python framework, since it was the simplest we could find and only depends on the Python standard library. We didn't need a database abstraction layer, support for Model View Controller pattern or any of the features that more advanced frameworks like Django offer. Bottle supports Hyper Text Markup Language (HTML) templating and provides a simple API for serving web content and that was enough for our needs. This let us write our entire web server in less than half a page of code, enabling us to focus most of our efforts on the client side which is where all the logic should be. In Figure 6.2) we can see the full code for our web server. When the user requests the web page the server checks the login information and serves the content. Since this is a web application it has only one web page.



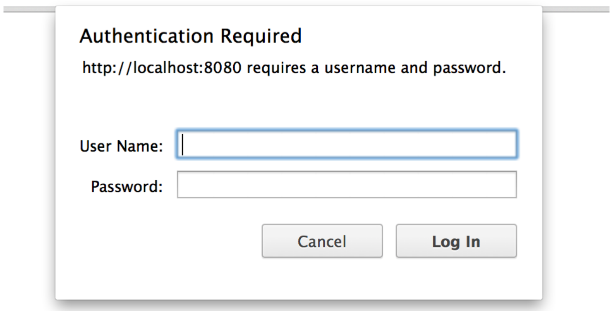
```python
from bottle import app, route, post, request, run, abort, template, static_file, url, auth_basic
import sqlite3

@route('/static/:path#.+#', name='static')
def static(path):
    return static_file(path, root='static')

def check_login(user, passw):
    conn = sqlite3.connect('arm_control_web.db')
    c = conn.cursor()
    c.execute("SELECT username FROM users WHERE username = '{}'".format(user))
    return c.fetchone() is not None

@route('/logout')
def logout():
    abort(401, "You're no longer logged in")

@route('/')
@auth_basic(check_login)
def home():
    return template('static/resources/tpl/home.tpl', get_url=url)

run(host='localhost', port=8080, debug=True)
```

Figure 6.2: Full Code for our Web Server in Bottle, performing authentication and serving static files

Because our web application does not need to be on the same machine as the back-end, we need a way to interact with it over HTTP. For this purpose we used Rosbridge [5] and Roslibjs [6]. Roslibjs is a JavaScript library that provides a JSON API to interact with ROS over HTTP by connecting with Rosbridge Server. Rosbridge Server is a ROS node that will run on the backend. This lets us publish and receive ROS messages between the client and server, which is all we needed to communicate with our backend node.

6.2 Database and Access

We implemented access control to make sure only robotics students can access the web application and control the robot. All usernames and passwords are stored in a database. When trying to access the web application it will ask the student for his credentials (see Figure 6.3) and check the database to validate that the credentials entered are correct.


Authentication Required

http://localhost:8080 requires a username and password.

User Name:

Password:

Figure 6.3: User Access Control

This database will just be used to persist information about users, so it will only have one table. Inserts will happen only once every year when enrollment for robotics course is finished and it will only be read to validate a student's login. Due to the low requirements of the database we chose SQLite because of its simplicity and speed. It doesn't require any configuration, it is self-contained and easily accessed through Python.

6.3 Design

We used Bootstrap to help us create the web application's design. Bootstrap is an HTML, Cascading Style Sheets (CSS) and Javascript framework created by Twitter that simplifies web UI design and ensures that it renders well on desktop and mobile. By using bootstrap components we obtain an appealing and consistent UI without having to manually tweak CSS.

Once we had the interface we had to add interaction. We considered AngularJS and React. AngularJS plays well with Bootstrap but we felt that for our use case it didn't give us enough advantages to justify including it as a resource. React was discarded after learning that it doesn't play well with Bootstrap. We finally decided that for our purposes it would be best to use plain Javascript and JQuery.

We had three different functionalities that we wanted to represent on our UI. The first was a text editor to write MB4 code, the second was a points editor to facilitate creating point coordinates for our program to use, and the third was the jog operation that would enable us to control the robot manually. To represent those separate functionalities we decided to use Bootstrap tab panes, which can be seen in figure 6.4. The advantage of this approach is that the functionalities don't have to share screen space, as only one is displayed at a time. This is not a problem because the user will usually use one at a time and will not need to change tabs very often. As a result, we have much more space to make each interface functional and easier to use. The following sections will explain each element of the interface.

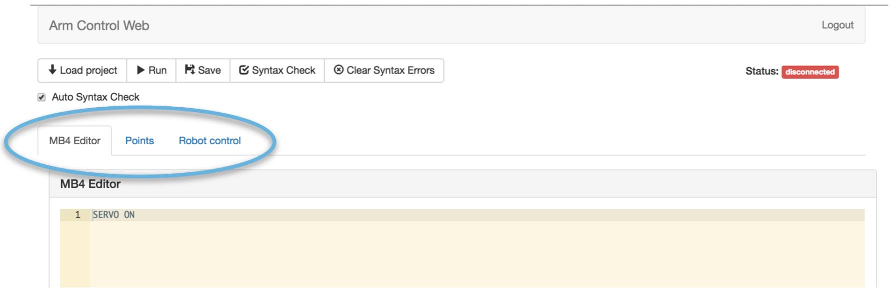


Figure 6.4: Web Interface with Tabs

6.4 Connection Status

At the top right corner, below the logout button there is a label that displays the connection status (see Figure 6.5 and 6.6). This label is red if the application is disconnected from the server and green if it is connected. We should note that this only represents if the application is connected to the Execute Node backend and has nothing to do with whether the backend is connected to the controller or not.

If the connection status is disconnected and we click on the label it will try to connect again.

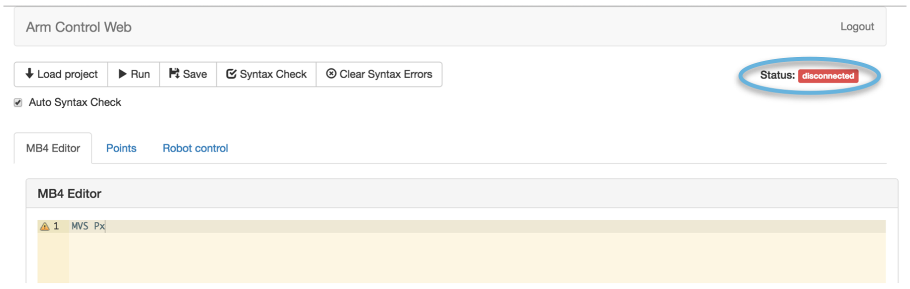
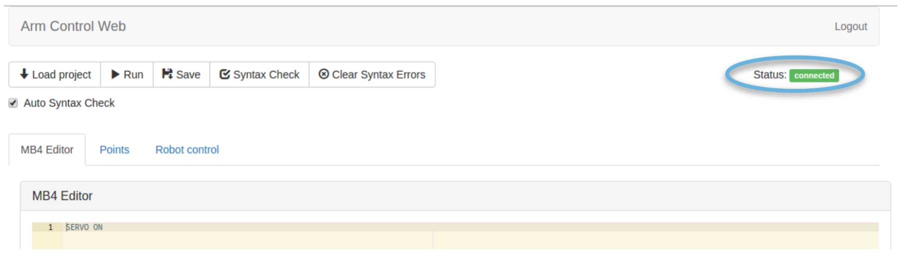


Figure 6.5: Application Disconnected from Backend

6.5 Code Editor

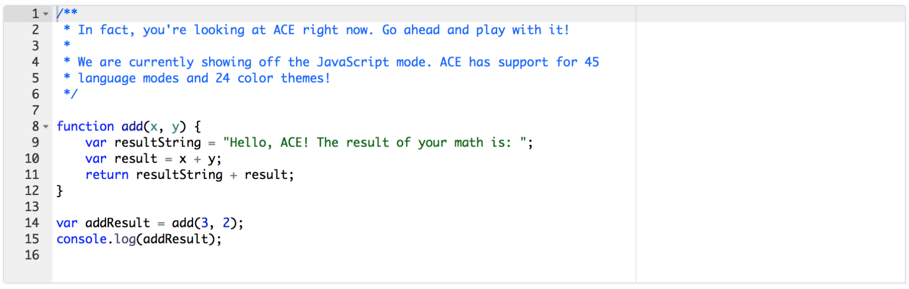
One of the most important parts to highlight in our interface was the code editor. It is where the students will spend most of their time writing and testing MB4 code for their assignments. Developing a good code editor is time consuming, hard and out of scope for this project, but since the web ecosystem is so mature we decided to look for an existing code editor that could be embedded in our website and provide the desired functionality.



```
SERVO ON
```

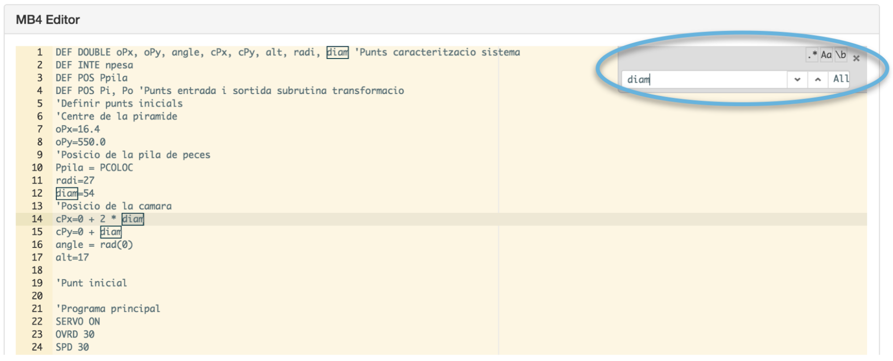
Figure 6.6: Application Connected to Backend

We finally decided that Ace, a code editor written and maintained by Cloud9 in Javascript, was the best option. It is a mature editor used in many successful websites such as Codecademy which can be easily embedded in any website and customized as needed. Figure 6.7 shows an example Ace Javascript editor. It can be extended with new modes and themes, and provides modes for all major languages as well as many of the features that are expected out of a code editor such as line numbering, search and replace (see Figure 6.8), highlighting matching parenthesis and automatic indentation.



```
/**
 * In fact, you're looking at ACE right now. Go ahead and play with it!
 *
 * We are currently showing off the JavaScript mode. ACE has support for 45
 * language modes and 24 color themes!
 */

function add(x, y) {
    var resultString = "Hello, ACE! The result of your math is: ";
    var result = x + y;
    return resultString + result;
}

var addResult = add(3, 2);
console.log(addResult);
```

Figure 6.7: Example Ace Editor



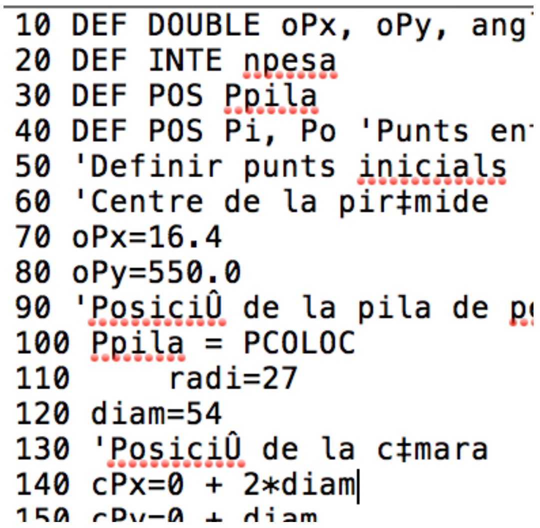
```
DEF DOUBLE oPx, oPy, angle, cPx, cPy, alt, radi, diam 'Punts caracteritzacio sistema
DEF INTE npesa
DEF POS Ppila
DEF POS Pi, Po 'Punts entrada i sortida subrutina transformacio
'Definir punts inicials
'Centre de la piramide
oPx=16.4
oPy=550.0
'Posicio de la pila de peces
Ppila = PCOLOC
radi=27
diam=54
'Posicio de la camara
cPx=0 + 2 * diam
cPy=0 + diam
angle = rad(0)
alt=17

'Punt inicial

'Programa principal
SERVO ON
OVRD 30
SPD 30
```

Figure 6.8: Ace Search

6.5.1 Automatic numbering

As we mentioned in Section 2.4, Melfa Basic IV requires the programmer to write a number at the start of each line. This line number can serve as a target for control flow statements such as goto and gosub statements. It also supports labels which are a better way to program since lines of code can more easily be added and removed without having to renumber them each time. This means that for programmers who are doing things correctly and only using labels as a target for their control flow statements, line numbers are not only redundant and unnecessary, but also an inconvenience. Every time code is added, removed or pasted, the lines need to be renumbered to make sure the program compiles correctly. This is very inconvenient and means that the process of refactoring or adding functionalities is not agile at all.

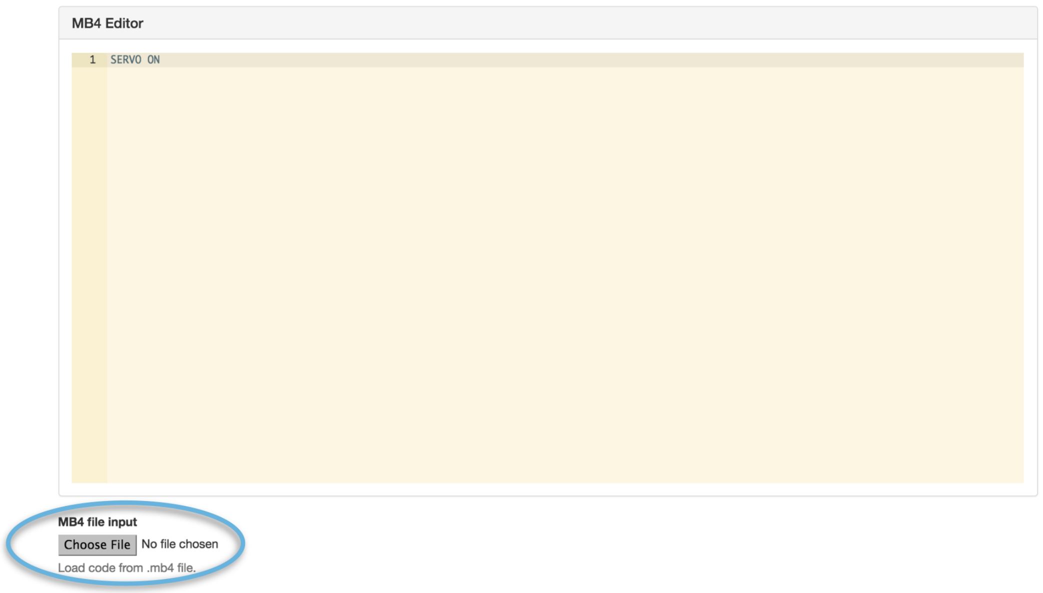
To solve this problem, our code editor expects the user to program without writing the line numbers at the start of each line. When the web application needs to send the code, the JavaScript function responsible for doing so will automatically add numbers at the start of each line. The number added will be the current line number multiplied by ten. Figure 6.9 shows the generated code with automatic numbering. This way we can program more conveniently while letting the editor do the tedious and unnecessary work for us. One possible caveat is that if code is loaded from an MB4 file the the original numbering will be lost and it will be replaced by the automatic numbering once it is sent to the backend. This should not be a problem if the user programs using labels.



```
10 DEF DOUBLE oPx, oPy, ang
20 DEF INTE npesa
30 DEF POS Ppila
40 DEF POS Pi, Po 'Punts en
50 'Definir punts inicials
60 'Centre de la pir‡mide
70 oPx=16.4
80 oPy=550.0
90 'PosiciÛ de la pila de p
100 Ppila = PCOLOC
110     radi=27
120 diam=54
130 'PosiciÛ de la c‡mara
140 cPx=0 + 2*diam
150 cPy=0 + diam
```

Figure 6.9: Melfa Basic IV code with automatically generated line numbers after saving from the Web Interface

Our interface provides a way to load a program from a local file (see Figure 6.10) and it automatically removes all line numbers. In Figure 6.11 we are selecting a file to load from our filesystem. Similarly, when the user chooses to save the file it adds the numbers back in automatically. If the file's extension is not .MB4 the user will be notified and the file will not be loaded.



```
SERVO ON
```

Figure 6.10: Button For Loading an MB4 File

6.6 Syntax checker

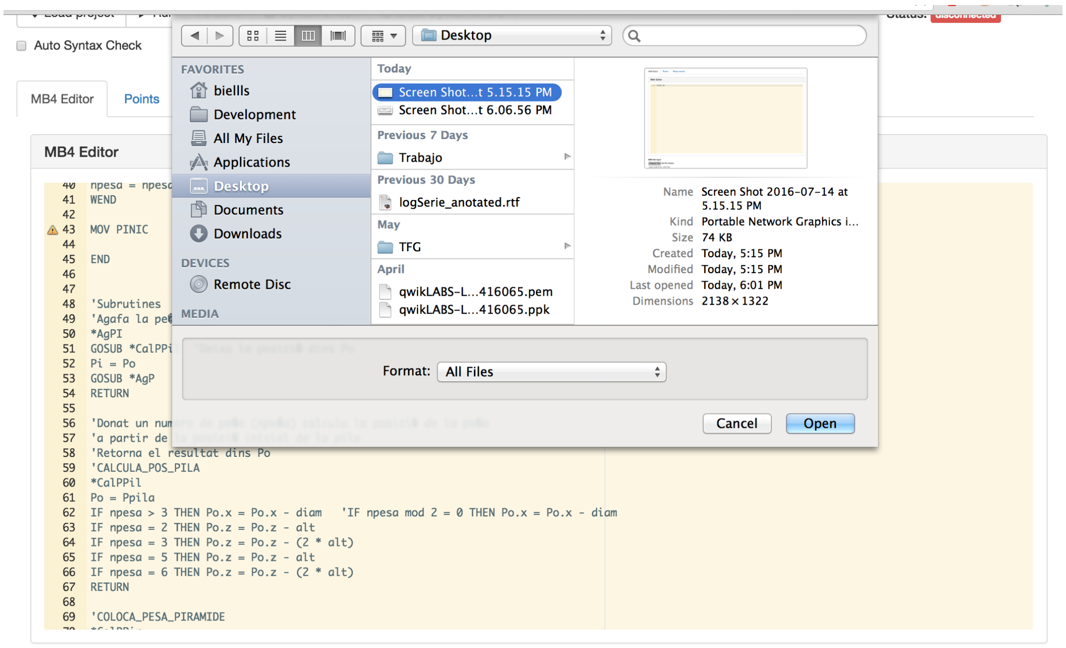
6.6.1 Introduction

When the user sends the program to the backend, and this in turn sends it to the controller, the controller will beep and be in an error state if there was an error in the code so that the robot will not attempt to run incorrect programs. This tells us that there is a problem in our code, but it does not give any information about what the problem might be or where it is located. Finding about errors in the code at run-time after having loaded it to the controller is at the very least extremely inconvenient. Furthermore, without hints about what or where the error might be, once the code is long enough it can be very challenging to find and correct the mistake.

In order to assist the programmer and avoid errors before sending the program, we propose a syntax checker which can immediately report errors and warnings in a large subset of Melfa Basic IV's grammar. In Figure 6.12 an error is reported in line 2 because we have written "INT" instead of "INTE".

Syntax checking is performed on the fly every time there is a change in the editor and errors are reported immediately on Ace's sidebar. This is achieved by manually passing the error lines and descriptions to Ace in the format that it expects.

Sometimes the user might not want syntax checking to happen automatically, so on the fly syntax checking can be turned off by deselecting the checkbox shown in Figure 6.13. The user can then manually tell the application to perform syntax checking when he is ready by pressing the Syntax Check button from the main buttons. Similarly, he can also press a button (also shown in Figure 6.13) to clear all syntax errors from the editor.

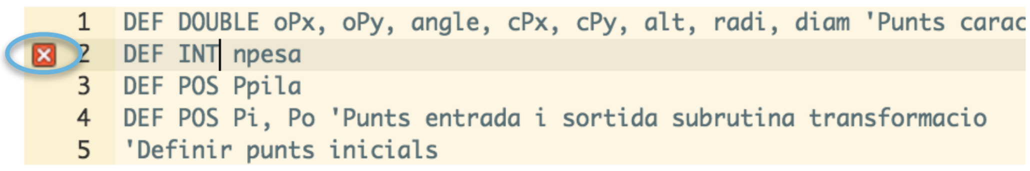


```
npesa = npesa
WEND

MOV PINIC

END


'Subrutines
'Agafa la pe
*AgPI
GOSUB *CalPPi
Pi = Po
GOSUB *AgP
RETURN

'Donat un num
'a partir de
'Retorna el resultat dins Po
'CALCULA_POS_PILA
*CalPPil
Po = Ppila
IF npesa > 3 THEN Po.x = Po.x - diam   'IF npesa mod 2 = 0 THEN Po.x = Po.x - diam
IF npesa = 2 THEN Po.z = Po.z - alt
IF npesa = 3 THEN Po.z = Po.z - (2 * alt)
IF npesa = 5 THEN Po.z = Po.z - alt
IF npesa = 6 THEN Po.z = Po.z - (2 * alt)
RETURN

'COLOCA_PESA_PIRAMIDE
```

Name Screen Shot 2016-07-14 at 5.15.15 PM
Kind Portable Network Graphics i...
Size 74 KB
Created Today, 5:15 PM
Modified Today, 5:15 PM
Last opened Today, 6:01 PM
Dimensions 2138 × 1322

Format: All Files

Figure 6.11: Dialog for choosing an MB4 File



```
DEF DOUBLE oPx, oPy, angle, cPx, cPy, alt, radi, diam 'Punts carac
DEF INT npesa
DEF POS Ppila
DEF POS Pi, Po 'Punts entrada i sortida subrutina transformacio
'Definir punts inicials
```

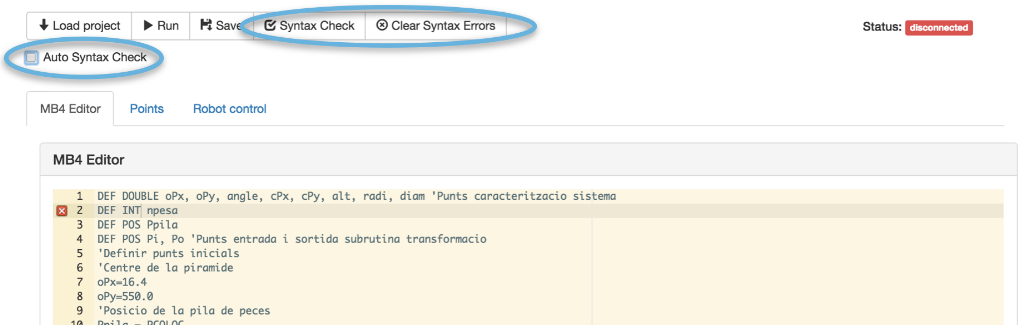
Figure 6.12: Syntax Error Reported by Ace

6.6.2 BNF MB4 Grammar

To be able to implement a lexer and parser first we had to define a grammar for our language. Since there is no official complete grammar specification for Melfa Basic IV, we had to create it ourselves based on the BASIC BNF grammar provided on Rosetta Code [7] and the Melfa Basic IV manual [8].

Below we provide part of our grammar specification for Melfa Basic IV in Extended BNF notation:

```
ID          =  {a–zA–Z}+
Real        =  {0–9}+\.{0–9}+
Integer     =  {0–9}+
String      =  "{Letter}*"
Boolean     =  TRUE
            |  FALSE
EOL         ::=  \n
             |  \r\n
```



```
DEF DOUBLE oPx, oPy, angle, cPx, cPy, alt, radi, diam 'Punts caracteritzacio sistema
DEF INT npesa
DEF POS Ppila
DEF POS Pi, Po 'Punts entrada i sortida subrutina transformacio
'Definir punts inicials
'Centre de la piramide
oPx=16.4
oPy=550.0
'Posicio de la pila de peces
```

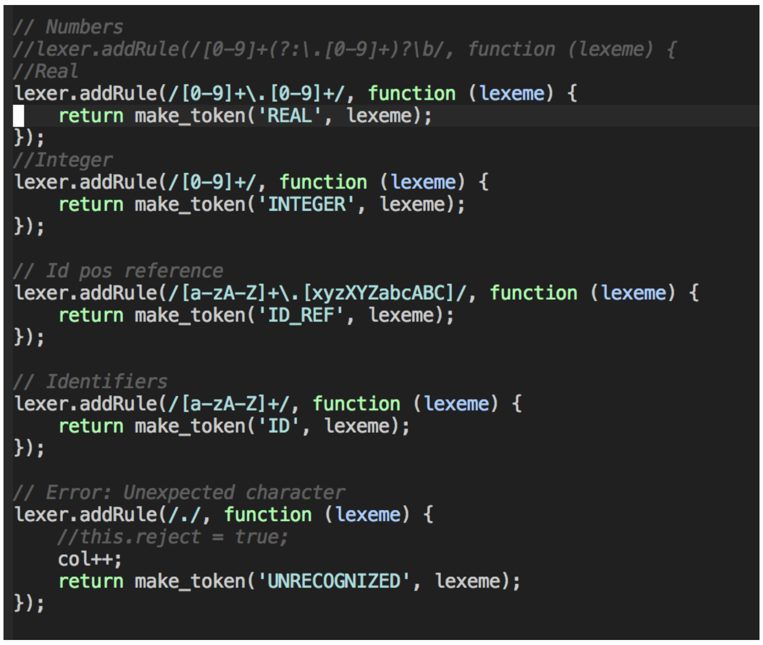
Figure 6.13: On the Fly Syntax Checking Disabled and Manual Controls

```
<PROGRAM>          ::= <STMTS>

<STMTS>            ::= <STMT> EOL <LINES>
                         | <STMT>

<STMT>             ::= DEF DOUBLE <ID_LIST>
                         | DEF CHAR <ID_LIST>
                         | DEF FLOAT <ID_LIST>
                         | DEF INTE <ID_LIST>
                         | DEF POS <ID_LIST>
                         | SERVO ON
                         | SERVO OFF
                         | <REF> = <EXP>
                         | GOTO Label
                         | GOSUB Label
                         | Label
                         | RETURN
                         | IF <EXP> THEN <STMT>
                         | WHILE <EXP> EOL
                           <STMTS>
                           WEND
                         | FOR ID = <EXP> TO <EXP> [STEP <EXP>] EOL
                           <STMTS>
                           NEXT [<ID_LIST>]
                         | SELECT <EXP> EOL
                           {CASE <EXP> EOL <STMT> BREAK EOL}+
                           [DEFAULT EOL <STMT> BREAK EOL]
                           END SELECT
                         | MOV ID [, [+-] Integer]
                         | MVS ID [, [+-] Integer]
                         | END
                         | DLY Integer
| HALT
| HOPEN Integer
| HCLOSE Integer
| JOVRD Integer
| OVRD Integer
| SPD Integer
```

A detailed specification of the complete grammar can be found in Section 9.3 of the Appendix.

6.6.3 Lexer

To implement our lexical analyzer we used the existing Lexer Javascript library. This library lets us add rules for lexical analysis in the form of Javascript regular expressions and return a token once a match is found. We will also return the line where the token was found, which will be useful to report errors. We can see some of our Lexer rules in Figure 6.14.



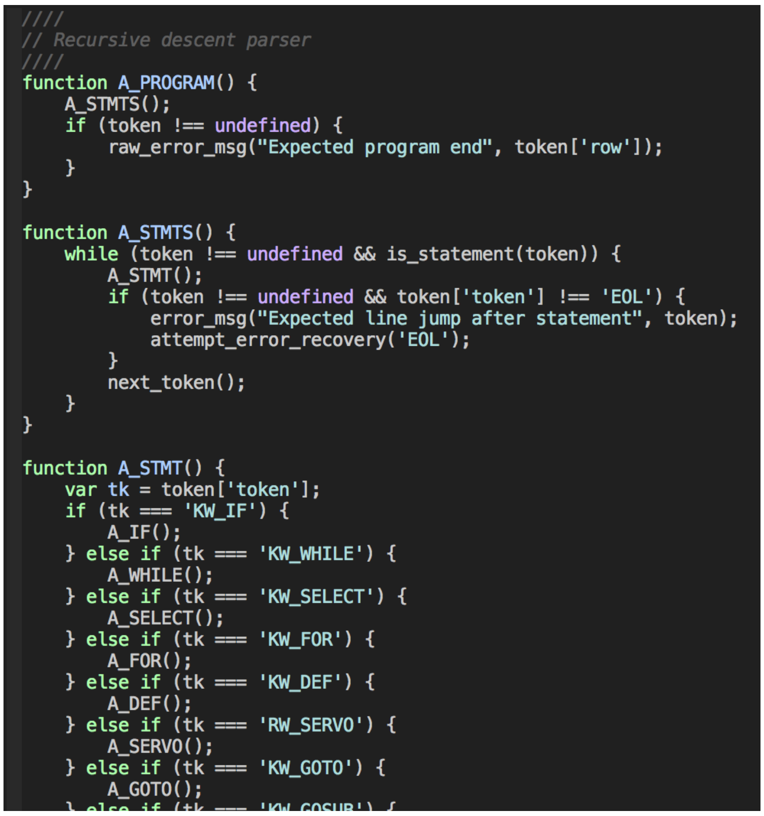
```
// Numbers
//lexer.addRule(/[0-9]+(?:\.[0-9]+)?\b/, function (lexeme) {
//Real
lexer.addRule(/[0-9]+\.[0-9]+/, function (lexeme) {
    return make_token('REAL', lexeme);
});
//Integer
lexer.addRule(/[0-9]+/, function (lexeme) {
    return make_token('INTEGER', lexeme);
});

// Id pos reference
lexer.addRule(/[a-zA-Z]+\.[xyzXYZabcABC]/, function (lexeme) {
    return make_token('ID_REF', lexeme);
});

// Identifiers
lexer.addRule(/[a-zA-Z]+/, function (lexeme) {
    return make_token('ID', lexeme);
});

// Error: Unexpected character
lexer.addRule(/./, function (lexeme) {
    //this.reject = true;
    col++;
    return make_token('UNRECOGNIZED', lexeme);
});
```

Figure 6.14: Some Lexer Rules

6.6.4 Recursive Descent Parser

To implement our parser we had several options. We gave consideration to the idea of using Jison, a Javascript library based on popular parsers like Bison and Yacc that given a context free grammar outputs a Javascript file that can parse it. The advantage of this approach is that it is much easier to create a grammar than to create a parser. It is also much easier to modify if the specification changes. However, there are disadvantages to this approach, one of them being that error messages will not be very clear. If Jison is given an incorrect program as input it will inform that there is an error at a certain line and column, but it will not be able to tell us much more. It would also be difficult to capture those errors to send them to Ace. Also, we are limited to the parsing done by Jison and cannot easily implement any static analysis or custom error recovery.

Due to those inconveniences, we decided that Melfa Basic IV was a small enough language that we could justify writing our own parser because the benefits outweighed the cost of writing it. We decided to implement a recursive descent parser because it is powerful enough to parse any language and is easy to create from the grammar specification. This would give us fine-grained control over every aspect of our parser to implement extensive error checking, recovery and to perform static analysis.



```
////
// Recursive descent parser
////
function A_PROGRAM() {
    A_STMTS();
    if (token !== undefined) {
        raw_error_msg("Expected program end", token['row']);
    }
}

function A_STMTS() {
    while (token !== undefined && is_statement(token)) {
        A_STMT();
        if (token !== undefined && token['token'] !== 'EOL') {
            error_msg("Expected line jump after statement", token);
            attempt_error_recovery('EOL');
        }
        next_token();
    }
}

function A_STMT() {
    var tk = token['token'];
    if (tk === 'KW_IF') {
        A_IF();
    } else if (tk === 'KW_WHILE') {
        A_WHILE();
    } else if (tk === 'KW_SELECT') {
        A_SELECT();
    } else if (tk === 'KW_FOR') {
        A_FOR();
    } else if (tk === 'KW_DEF') {
        A_DEF();
    } else if (tk === 'RW_SERVO') {
        A_SERVO();
    } else if (tk === 'KW_GOTO') {
        A_GOTO();
    } else if (tk === 'KW_GOSUB') {
```

Figure 6.15: Recursive Descent Parser Code Sample

At every step of the parser we check that the current token is what we expect it to be. If it is not we try to output an error message that is as descriptive as possible. For example, given the following program where we have omitted the THEN keyword:

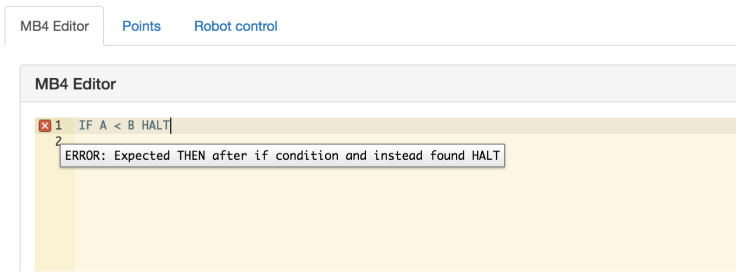
```
IF A < B
```

We would get the error message "ERROR: Expected THEN after if condition and instead found HALT" as is shown in Figure 6.16. Let's take a look at another example where the asterisk before a label name is missing:

```
GOTO *mylabel
mylabel
```

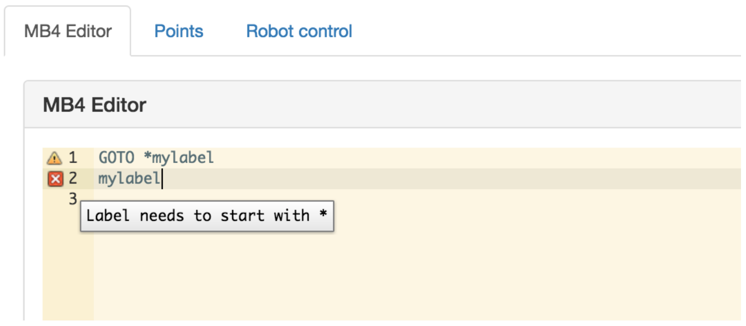
We would get the error message "Label needs to start with *" on line two, which tells us not only where the error is located but also why it happened and how to fix it. We can see this error in Figure 6.17.

This level of specificity of the error messages is only possible because we have created the parser manually and can contemplate and treat every case individually.



```
IF A < B HALT
ERROR: Expected THEN after if condition and instead found HALT
```

Figure 6.16: Example Error Message



```
GOTO *mylabel
mylabel
Label needs to start with *
```

Figure 6.17: Another Example Error Message

This kind of error checking produces detailed and informative errors that can help programmers produce code fast and with few errors, but there is one case where the recursive descent parser fails to produce good messages. When we try to write a statement and we make a mistake on the first keyword, for example typing "WIHLE" instead of "WHILE", the parser would interpret it as an identifier instead of a keyword so it wouldn't call the function to check a while loop and as a result it wouldn't check the rest of the statement correctly. We cannot always guess what the user is trying to type if he makes a mistake, so there is no way to attempt to parse the rest of a while loop statement if he spelled while incorrectly. What we can do is inform the user that he entered an invalid command and try to guess what he was trying to write. That way after he corrects it the parser will be able to check the rest of the statement.

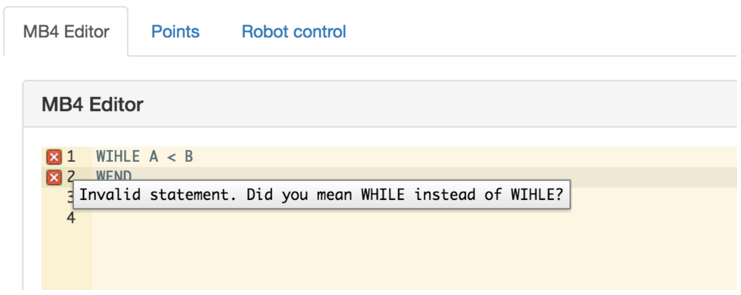
Since we should never find an identifier at the start of a statement except when there is an assignment, if we find one that is not followed by the '=' sign we try to see if the user might have misspelled a keyword. If this is not the case we assume that he was trying to write an assignment and inform that we expected an equals sign. The way we do this is by using the Levenshtein distance [9] to calculate the edit distance of the given identifier with every keyword that would be valid as a start of a statement.

We take the best match and if it is close enough to the given identifier we output an error message informing the user that a typo might have been made and suggesting the correct keyword.

Below we provide an example of how this would work. Notice we have misspelled "WHILE" as "WIHLE":

```
WIHLE A < B
WEND
```

In Figure 6.18 we can see that the error message provided is: "Invalid statement. Did you mean WHILE instead of WIHLE?." This works even if the words don't start with the same letter or are not the same length as long as the word the user typed is reasonably close to the correct one. For example "KOSUB" and even "KOSU" would be interpreted as a typo of "GOSUB," but "KOKUB" would be too different from "GOSUB" to accept it as a suggestion.



```
1  WIHLE A < B
2  WEND
3  Invalid statement. Did you mean WHILE instead of WIHLE?
4
```

Figure 6.18: Error Message Trying to Guess what the User wanted to Write using the Levenshtein Edit Distance

The pseudocode of our typo suggestion implementation can be found in Section 9.4 of the Appendix.

Error Recovery

As we discussed in previous sections, one of the advantages of creating our own parser is that we can perform custom error recovery. If we stopped and exited after finding the first error we might miss many more, so we need a way to keep going after reporting an error.

A good approach is to look for a specific token that we know should be there, skipping any tokens that we find on the way and continue error checking once we find the expected token. This ensures that we recover from the error and continue from a consistent state. This approach is very simple as we illustrate in the following pseudocode:

```
function attempt_error_recovery(goal_tk)
  tk = next_token()
  //Continue until we find the goal token or the end of file
  while tk != EOF and tk != goal_tk
    tk = next_token()
  end while
  return tk != EOF
```

From the above pseudocode we can see that the drawback of this approach is that if we don't find the goal token we will skip the whole file without performing any further error checking. This could have the unintended consequence of making us more conservative in choosing a token for error recovery. Observe the following program:

```
IF a < b HOPEN
IF end < start THEN HOPEN
```

Both "IF" statements contain multiple errors. The first "IF" statement is missing the keyword "THEN" and is also missing an integer after "HOPEN", for example "HOPEN 1." The second "IF" statement contains an error because "END" is a keyword so the condition expression is invalid. It is also missing an integer after "HOPEN."

With the above-mentioned pseudo-code approach there is no ideal answer as to which token we should choose as a goal token for error recovery. Suppose we took the conservative approach and chose the "EOL" end of line token. In this case we would detect the first error in the first statement, skip to the end of line, continue checking the second "IF" statement where we would find the first error, skip to the end of line and end the error checking missing two errors in total. The first error that we missed was very complicated to check, but could we have found the second one if we had chosen the 'THEN' keyword as a goal token? The answer is yes we could, but we would miss one of the errors we had previously detected in the process. After finding the first error on the first statement we would skip tokens until we found the "WHILE" keyword from the second "IF". We would then continue error checking finding an additional error in "HOPEN," but we would have already missed two errors in the process.

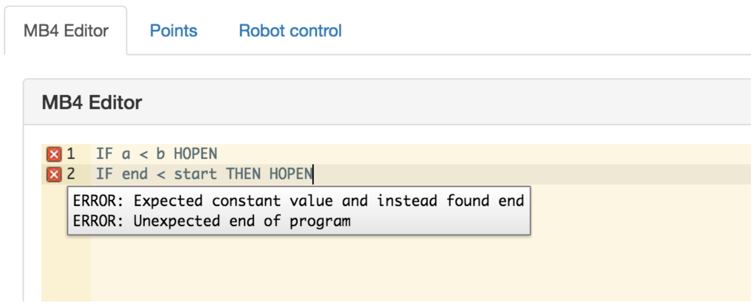
There is a way to improve on this. Once we realize that an "IF" statement should be all in one line, it is easy to see that we should only attempt to find the "THEN" keyword as long as we are still on the same line. Thus we can modify our pseudo-code of the attempt error recovery function to take a list of goal tokens instead of a single one. This way we will stop and continue error checking when we find any of the token in the list of goal tokens, whichever one comes first. Below the modified pseudo-code can be seen:

```
function attempt_error_recovery(goal_tks)
  tk = next_token()
  //Continue until we find a token that belongs to the list of
  //goal tokens or the end of file
  while tk != EOF and tk not in goal_tks
    tk = next_token()
  end while
  return tk != EOF
```

This simple change would allow us to find three out of four errors in the previous example, which is good enough for our purpose, as Figure 6.19.

Static Analysis

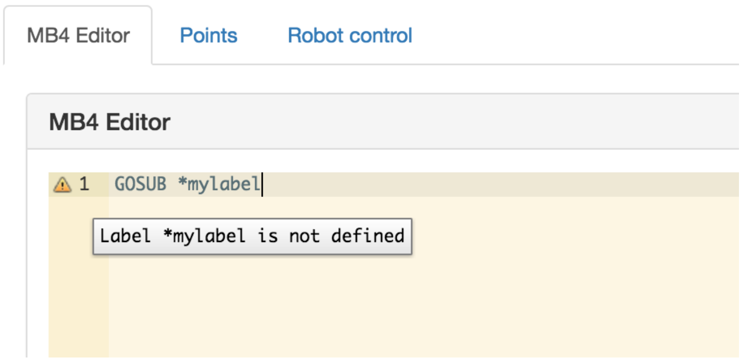
Now that we have done our best to report errors in a helpful way we can focus on performing some simple static analysis to give the user warnings if undefined points or labels are used. To do this, before starting to parse the code, we simply pattern match



```
IF a < b HOPEN
IF end < start THEN HOPEN
ERROR: Expected constant value and instead found end
ERROR: Unexpected end of program
```

Figure 6.19: Example of Error Recovery Where we Continue Reporting Errors After THEN Token

all labels that are located at the start of a line and store them in a list. When we are parsing a file and find a GOTO os GOSUB statement, we check that the label that it references belongs to the list of defined labels. If it does not, we report this as a warning that the label is undefined. Similarly, we also store all points defined in the points tab on a list and whenever we find a MOV or MVS during parsing we check that the point referenced belongs to that list. We produce a warning if it doesn't. In Figure 6.20 and Figure 6.21 we can see the respective warnings when a label and a point is not defined.



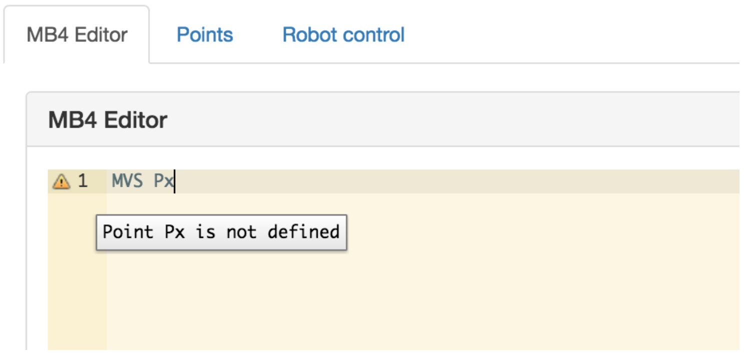
```
GOSUB *mylabel
Label *mylabel is not defined
```

Figure 6.20: Label not Defined Warning

6.7 Main Control Buttons

In our interface we have a main panel with control buttons to perform the most frequent actions. This panel, which can be seen in Figure 6.22, is located above the tab view, so it is always visible and can be accessed from any tab. It has a button for each of the following actions: loading the current project, running the currently loaded project, saving the current program or points file, syntax checking and clearing syntax errors.

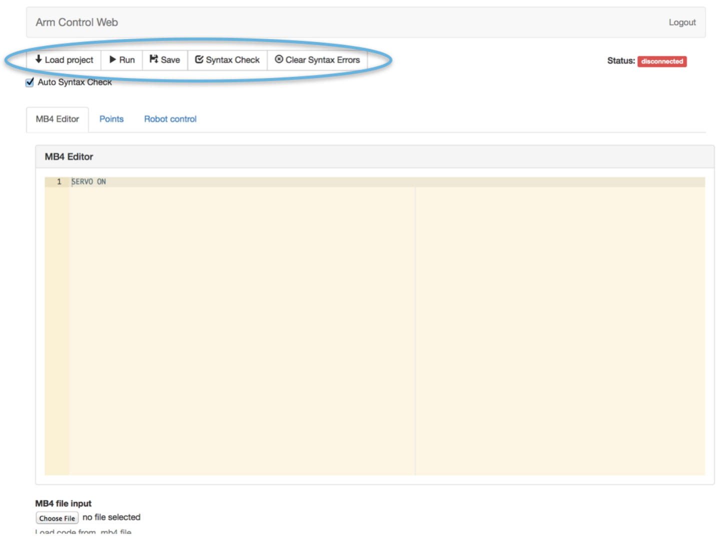
When the load project button is pressed, the web application first commands the backend to delete whatever is currently loaded in Slot 1, and then sends the code that is currently in the editor over Roslibjs. After that it also sends the points defined in the points tab.



```
MVS Px
```

Point Px is not defined

Figure 6.21: Point not Defined Warning. Will go Away Once the User Defines a Point Named Px in the Points Tab



```
SERVO ON
```

Figure 6.22: Main Control Buttons Panel

When we press the run button we ask the backend to tell the controller to run whichever program is currently loaded on the controller.

When we press the save button we get a dialog asking us to save the file to a path in our local filesystem. The save button is context dependent, so if it is pressed while on the MB4 editor tab it will save the editor contents into an .MB4 file. If it is pressed while on the load Points tab it will save the specified points into a .POS file, and if it is pressed it while on the robot control tab it will do nothing.

When we press the syntax check we are manually telling the application to run a syntax check and display the errors on the code editor. This is only useful when the "auto syntax check" checkbox is unchecked. To remove the errors from the editor we would press the "clear syntax errors" button.

6.8 Point Definitions

Any non-trivial Melfa Basic IV program requires the robot to move through space. This is done by calling MOV or MVS operations on defined points. However, these points are not defined in the .MB4 file, they are defined separately. In COSIROP a point can be defined by manually entering the values or by importing the current position from the jog operation. We wanted to match this functionality at the very least. Since point definitions are separate from the program we decided to include this functionality in a separate tab in our interface to illustrate that both concepts are separate. This also leaves more space to implement our UI. The points definition tab can be seen in Figure 6.23.

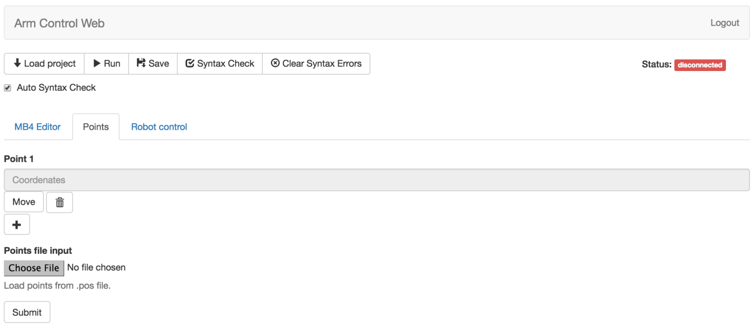


Figure 6.23: Point Definitions Tab

The first time the user accesses the tab there is a single empty point created. A message informing the user to click the point will appear if the user hovers his mouse over the point, as we show in Figure 6.24. If the user clicks it, a bootstrap modal window where all of the point's values can be set manually will appear.

After the user clicks the Add button, the new point will be automatically added in the form that is expected in a .POS file (explained in Section 2.4), which is the following:

```
PointName=(x,y,z,roll,pitch,yaw)(0,7)
```

The user never has to worry about the details of this syntax as he never edits the text directly, only through the modal form. To add more points the user can simply press the '+' button shown on figure 6.28 and a new entry will appear in the interface so that another point can be defined. Figure 6.25 shows the interface after a point named "Px" has been defined.

Although the interface needs to allow the user specify points manually, this is usually tedious and prone to errors. An alternative way to define a point is by moving the robot to the desired position and pressing the "Import Current Robot Position" button in the modal form. This will tell the backend to ask the controller for the current tool pose of the robot. Once it receives the position it will publish it as a ROS message so that the web interface can access it through Roslibjs. The point coordenates will then appear automatically on the modal window where the user will only have to specify a name for

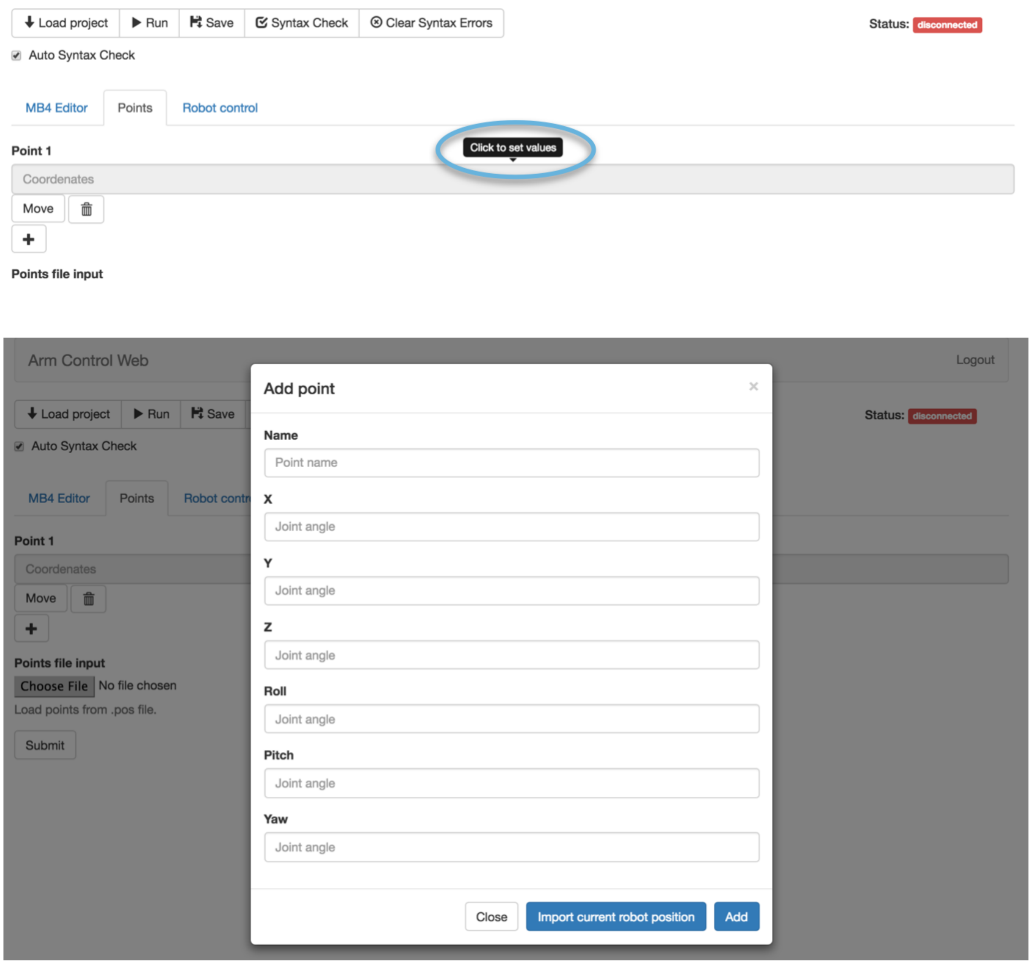


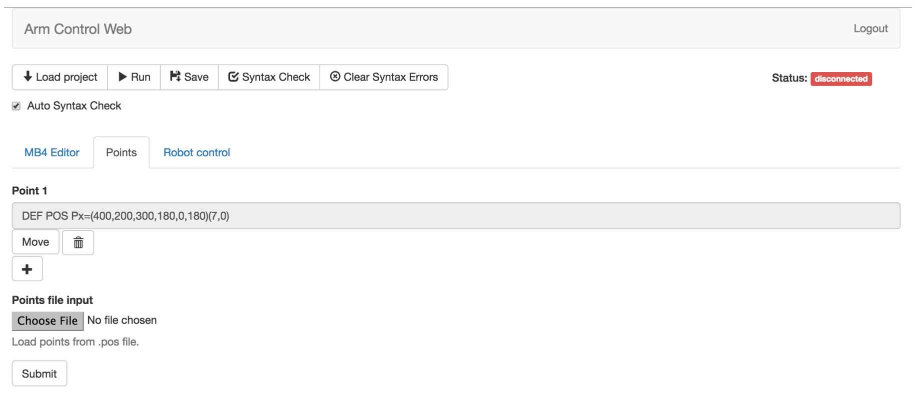
Figure 6.24: Click the Point to Set Values

the point before he can save it. Once the user has finished adding points he can click the save button to save the defined points to a .POS file (see figure 6.28).

Finally, as we can see in Figure 6.29, the user can also load points from a file. He is required to choose a file with a .POS extension or the file will not be loaded and an error will be shown. When the file loads it will overwrite any previously defined points and it will create new point entries if necessary.

6.9 Jog Operation

In addition to programming the robot and setting points we also need to be able to control it manually. This can be achieved with the teaching pendant, which is the fastest and most accurate way to control the robot since it is connected directly to the controller. However, as we explained in Section 2.2, to be able to control the robot with the teaching pendant we need to turn the controller key to "Teach" mode. This disconnects the controller from out program, which needs the key to be in "Auto" mode.



DEF POS Px=(400,200,300,180,0,180)(7,0)

Figure 6.25: Px point defined

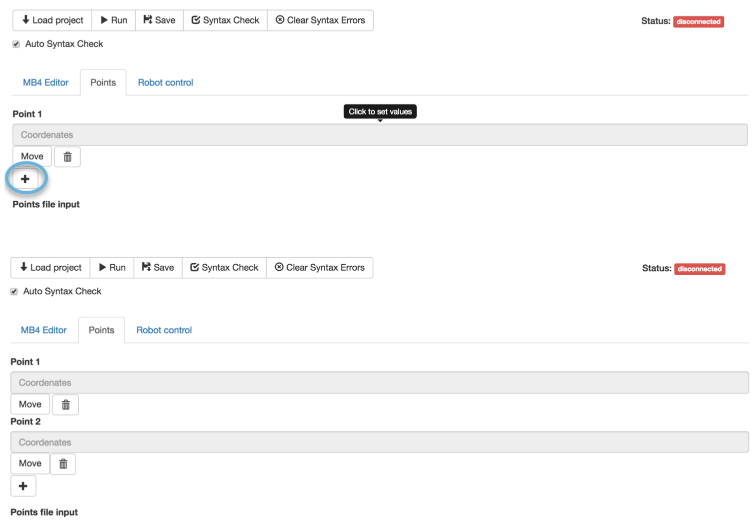


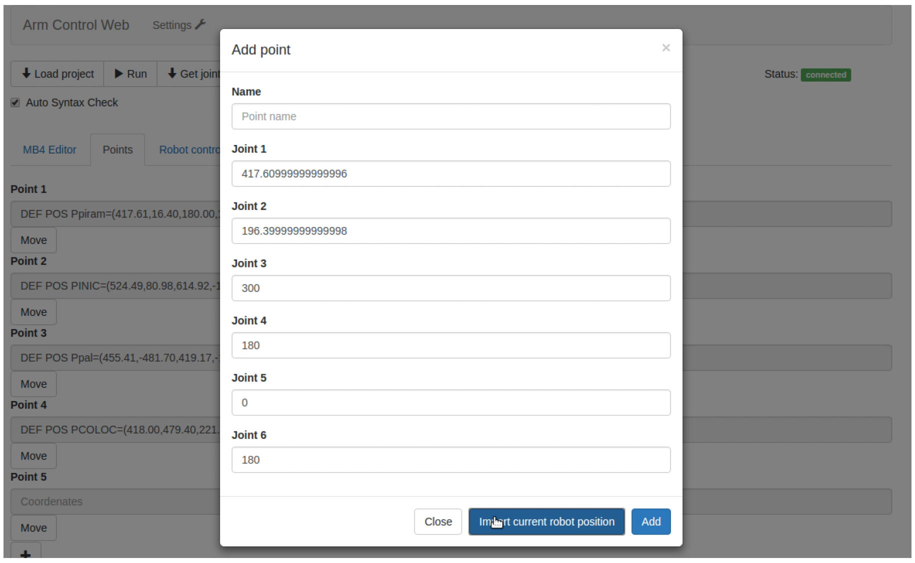
Figure 6.26: Add a New Point

For small and frequent robot adjustments this workflow might be time consuming and frustrating, which is why we introduced the "Robot Control" tab.

6.9.1 Control Scheme Implementation

We first considered implementing the jog operation with normal HTML buttons so it would be similar to the real teaching pendant and COSIROP's jog operation. This approach would have been simple to program, but we felt that having to use the mouse to move across the screen and press different buttons wasn't the most natural or efficient way.

Since this interface will run on a PC and we want to make it natural and comfortable to control the robot, we decided to take a hint from control schemes in computer games and use the keyboard for movements. Computer games are created with the sole purpose of being usable and fun to control, so we hypothesized that borrowing their


Arm Control Web Settings

Load project | Run | Get joint

Auto Syntax Check

MB4 Editor | Points | Robot contro

Point 1
DEF POS Ppiram=(417.61,16.40,180.00,
Move
Point 2
DEF POS PINIC=(524.49,80.98,614.92,-1
Move
Point 3
DEF POS Ppal=(455.41,-481.70,419.17,-
Move
Point 4
DEF POS PCOLOC=(418.00,479.40,221.
Move
Point 5
Coordenates
Move

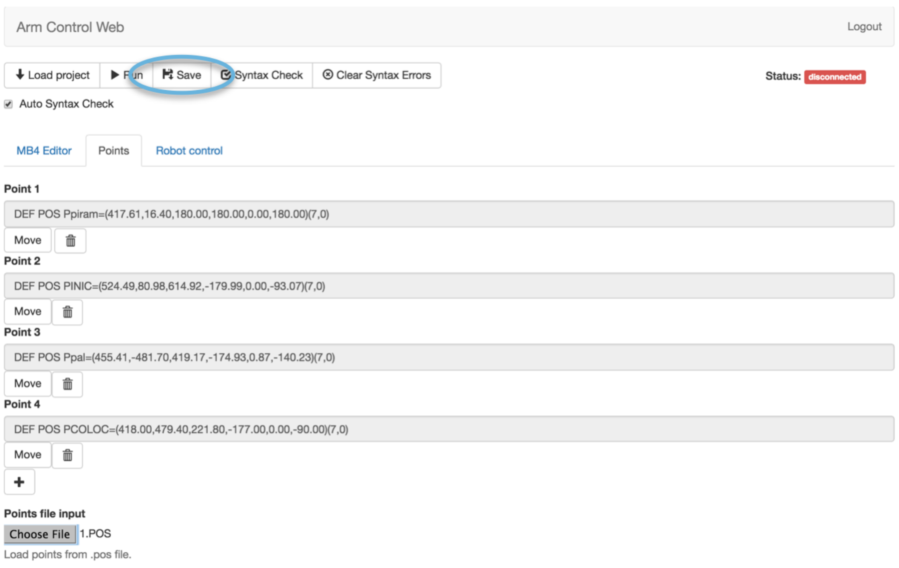
Status: connected

Add point

Name: Point name
Joint 1: 417.60999999999996
Joint 2: 196.39999999999998
Joint 3: 300
Joint 4: 180
Joint 5: 0
Joint 6: 180

Close | Import current robot position | Add

Figure 6.27: Import Point from Current Robot Position


Arm Control Web Logout

Load project | Run | Save | Syntax Check | Clear Syntax Errors

Status: disconnected

Auto Syntax Check

MB4 Editor | Points | Robot control

Point 1
DEF POS Ppiram=(417.61,16.40,180.00,180.00,0.00,180.00)(7,0)
Move
Point 2
DEF POS PINIC=(524.49,80.98,614.92,-179.99,0.00,-93.07)(7,0)
Move
Point 3
DEF POS Ppal=(455.41,-481.70,419.17,-174.93,0.87,-140.23)(7,0)
Move
Point 4
DEF POS PCOLOC=(418.00,479.40,221.80,-177.00,0.00,-90.00)(7,0)
Move

Points file input
Choose File 1.POS
Load points from .pos file.

Figure 6.28: Save Points to a .POS file

controls would improve our system's usability. We decided to use the classical "wasd" control scheme where "w" moves the robot forward, "a" moves it to the left, "s" moves it backwards and "d" moves it to the right. In addition to those four movements the robot also needed to be able to move upwards and downwards. We decided that assigning those actions to the up and down keys respectively would be appropriate. The last operation that we wanted to support was opening and closing the robot actuator, which we assigned to the space bar key. A full list of key-bindings is provided in Table 6.1.

The "Robot Control" tab is composed entirely of an HTML5 canvas that displays some images of keyboard keys (see Figure 6.30). When the focus is on the canvas

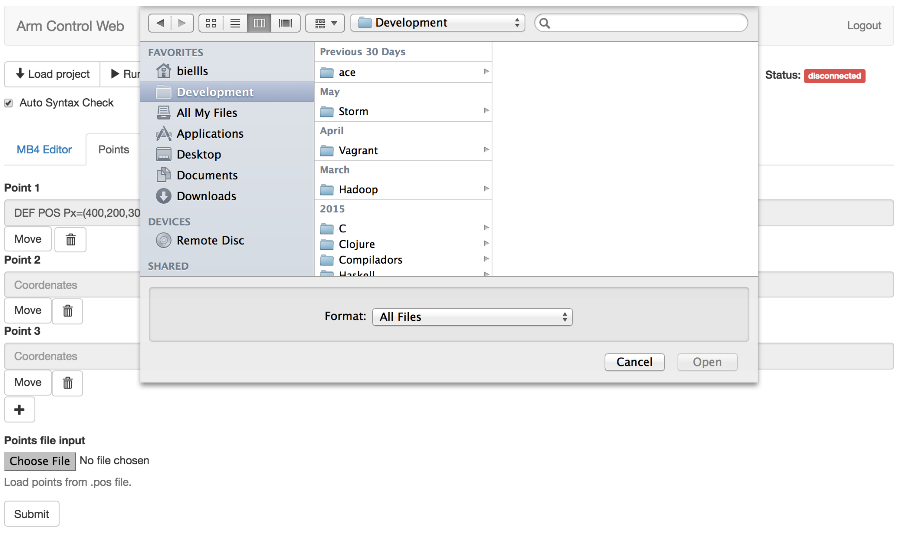


Figure 6.29: File Chooser to Load a Points File

Key	Action
w	Move tool in the +x direction
a	Move tool in the +y direction
s	Move tool in the -x direction
d	Move tool in the -y direction
up arrow	Move tool in the +z direction
down arrow	Move tool in the -z direction
spacebar	Open or close robot actuator

Table 6.1: Control Scheme showing Keys and their Corresponding Action

it captures all keyboard events and processes them by sending the necessary ROS messages to tell the backend to move the robot tool. Each displayed key represents a different jog operation. When the user presses a key, the corresponding key image in the canvas changes color (see Figure 6.31). This gives the user feedback that the key press has been detected and processed.

6.9.2 Actuator State

In the previous subsection we mentioned that the space bar is used to both open and close the robot tool. To the robot controller those are two different operations, so to be able to use them with just one key we need to keep some state. We will assume that the robot actuator is initially open. If the space bar is pressed we will tell the backend to close the actuator and assume that the actuator is now closed. If the space bar is pressed again we will tell the backend to close the actuator and assume that it is now open.

One minor disadvantage of this approach is that if the actuator was initially closed we would need to press the spacebar twice in order to open it, since on the first try we would attempt to close it and it would do nothing since it was already closed. Since we

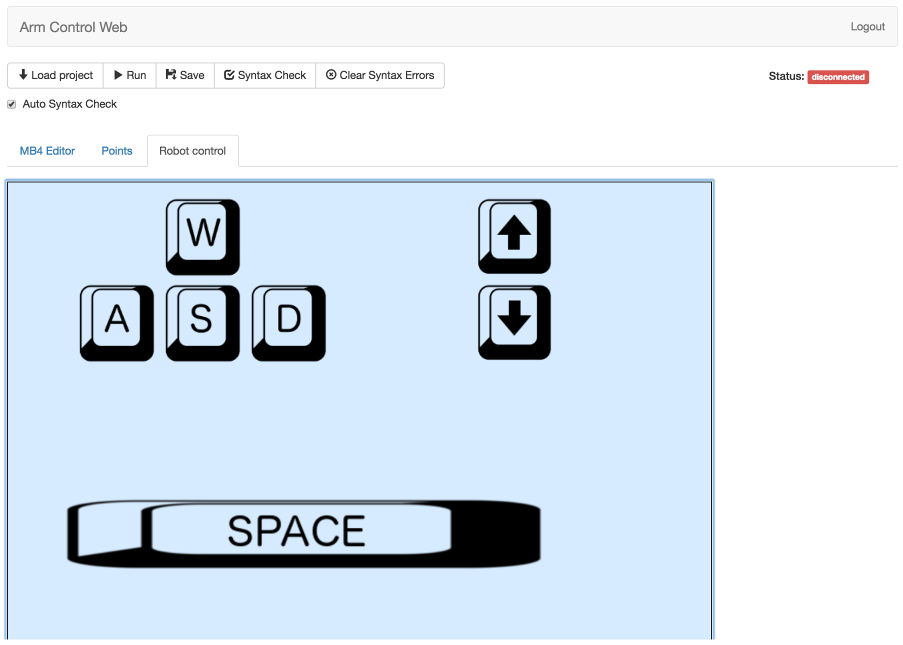


Figure 6.30: Robot Control HTML5 Canvas Displaying all Control Keys

cannot obtain the actuator state in the first place we decided that this compromise was worth it to avoid having two separate keys, one for opening and the other for closing.

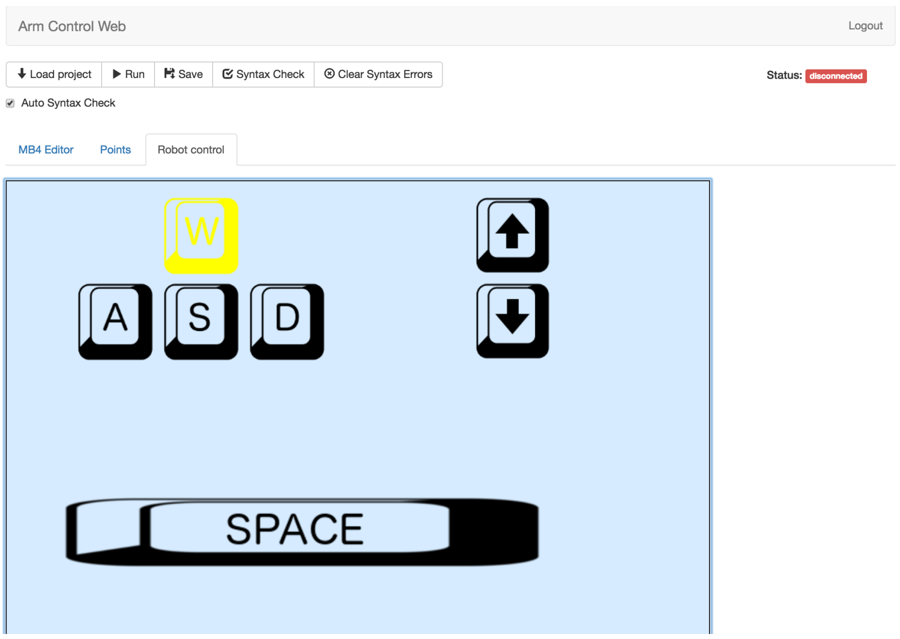


Figure 6.31: Robot Control HTML5 Canvas Showing that the "w" key is being pressed

Chapter 7

Tests

In this chapter we show some tests that help verify the correct functioning of each module.

7.1 Backend

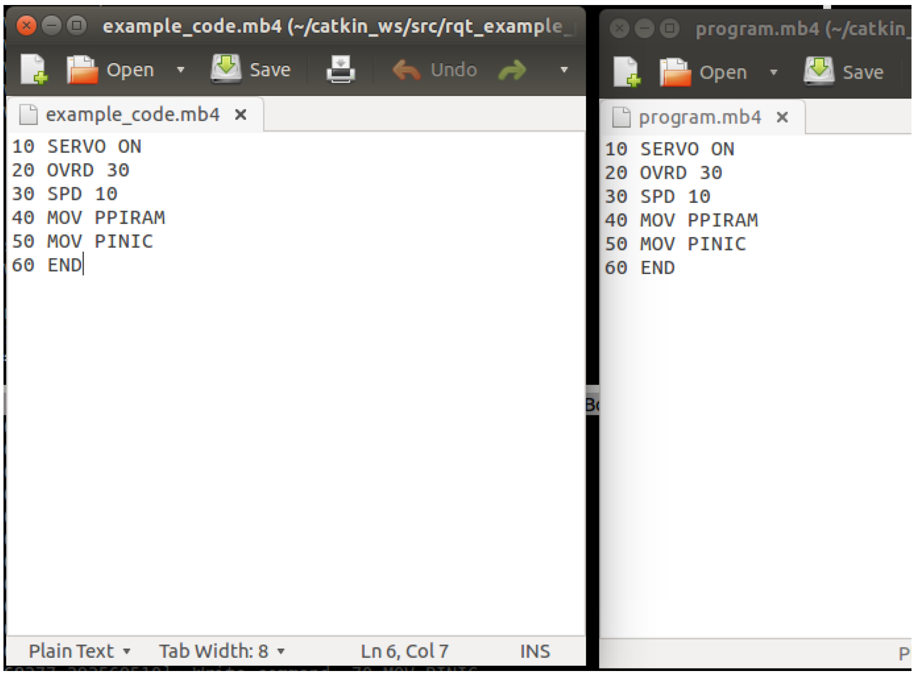
Below we detail some of the tests carried out to show the correctness of our backend platform.

7.1.1 Sending Programs and Points

1. The following Melfa Basic IV file, which fits in one message, was reconstructed correctly by the backend from the ROS messages it received. This was validated manually.

   ```
   10 SERVO ON
   20 OVRD 30
   30 SPD 10
   40 MOV PPIRAM
   50 MOV PINIC
   60 END
   ```

   In the following screen capture we can see that the file we sent and the file that was reconstructed by the backend are identical:



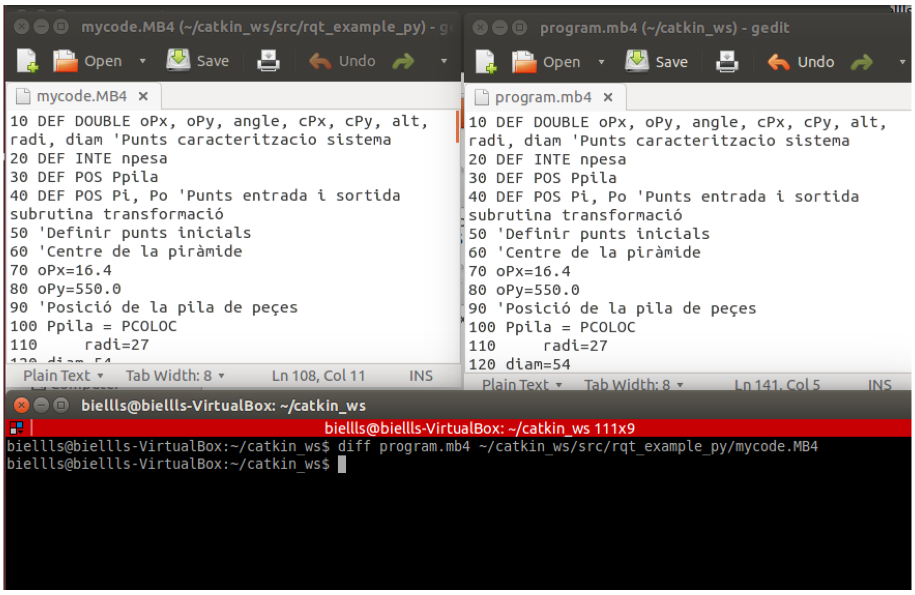
example_code.mb4 (~/catkin_ws/src/rqt_example_

```
10 SERVO ON
20 OVRD 30
30 SPD 10
40 MOV PPIRAM
50 MOV PINIC
60 END
```

program.mb4 (~/catkin_

```
10 SERVO ON
20 OVRD 30
30 SPD 10
40 MOV PPIRAM
50 MOV PINIC
60 END
```

2. A Melfa Basic IV file which was too long to fit in one message was reconstructed correctly by the backend. Due to the length of the file it is difficult to see if the file has been correctly received, so it was validated using the Linux diff command as can be seen below:



mycode.MB4 (~/catkin_ws/src/rqt_example_py) - g

```
10 DEF DOUBLE oPx, oPy, angle, cPx, cPy, alt,
radi, diam 'Punts caracteritzacio sistema
20 DEF INTE npesa
30 DEF POS Ppila
40 DEF POS Pi, Po 'Punts entrada i sortida
subrutina transformació
50 'Definir punts inicials
60 'Centre de la piràmide
70 oPx=16.4
80 oPy=550.0
90 'Posició de la pila de peçes
100 Ppila = PCOLOC
110     radi=27
```

program.mb4 (~/catkin_ws) - gedit

```
10 DEF DOUBLE oPx, oPy, angle, cPx, cPy, alt,
radi, diam 'Punts caracteritzacio sistema
20 DEF INTE npesa
30 DEF POS Ppila
40 DEF POS Pi, Po 'Punts entrada i sortida
subrutina transformació
50 'Definir punts inicials
60 'Centre de la piràmide
70 oPx=16.4
80 oPy=550.0
90 'Posició de la pila de peçes
100 Ppila = PCOLOC
110     radi=27
120 diam=54
```

```
biellls@biellls-VirtualBox:~/catkin_ws$ diff program.mb4 ~/catkin_ws/src/rqt_example_py/mycode.MB4
biellls@biellls-VirtualBox:~/catkin_ws$ 
```

3. The program and points files were loaded correctly into the controller. This was validated by extracting the files from the controller using COSIROP and checking that it was the same file.

4. A ROS message was manually sent for every possible movement command: move in the positive x, y or z direction, move in the positive x, y or z direction, open the

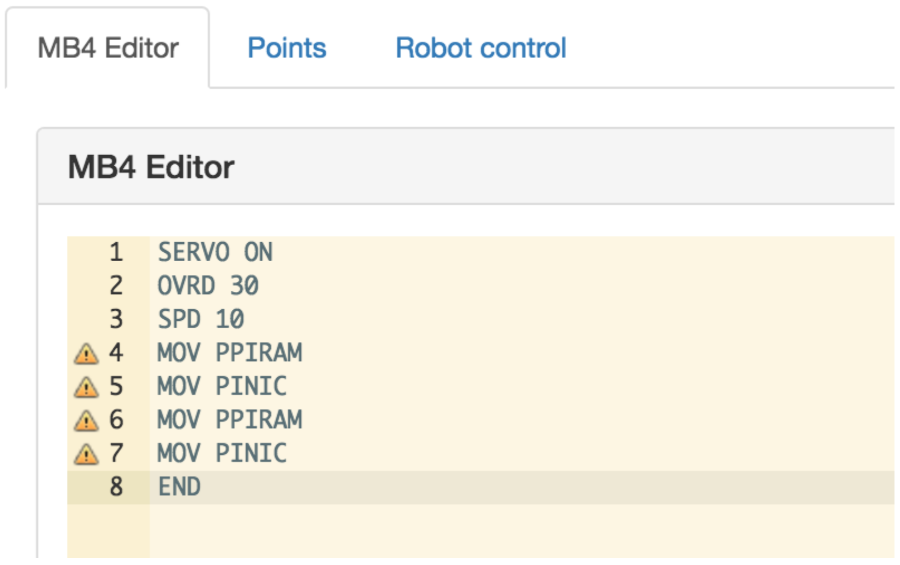
actuator or close the actuator. Below is an example ROS message sent through the command line. The robot servo remained on throughout the process and turned off when a non-movement command was sent to the backend, which served to validate that the jog operation thread was initialized, handled the messages correctly and was destroyed as expected.

```
rostopic pub -1 /execute_instruction std_msgs/String -- '---MOV TOOL +X---'
```

7.2 Frontend

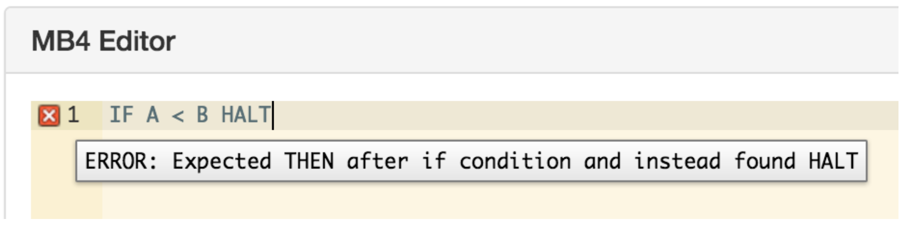
This section will detail some of the tests performed on the backend.

1. We loaded the same MB4 file from the previous section and manually validated that it was loaded without errors and that all line numbers had been removed from the start of the line.



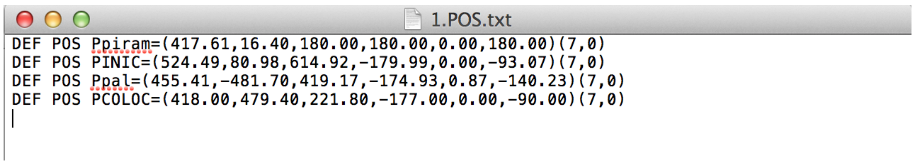
```
SERVO ON
OVRD 30
SPD 10
MOV PPIRAM
MOV PINIC
MOV PPIRAM
MOV PINIC
END
```

2. Every MB4 statement was tested extensively in the code editor to make sure the expected error was produced and that it was sufficiently descriptive.

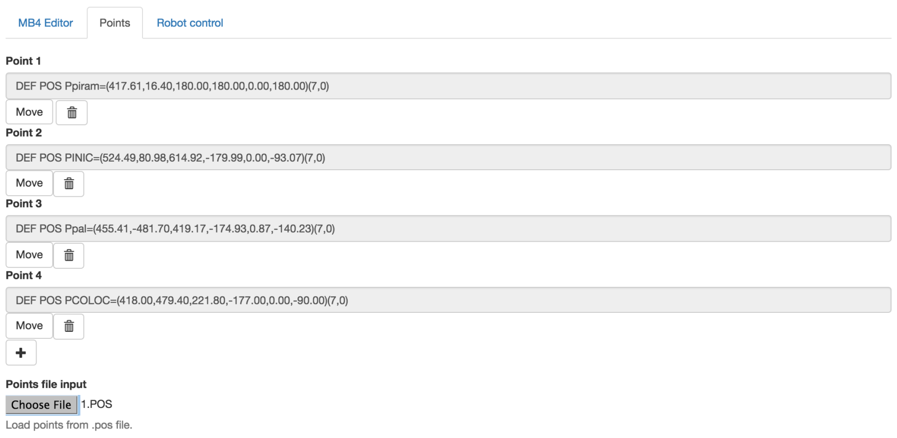


```
IF A < B HALT
ERROR: Expected THEN after if condition and instead found HALT
```

3. We validated that the .POS file shown below was loaded correctly into the application. Below we can see the original text file containing the positions, and the corresponding points that were created in the interface when this text file was loaded.



```
DEF POS Ppiram=(417.61,16.40,180.00,180.00,0.00,180.00)(7,0)
DEF POS PINIC=(524.49,80.98,614.92,-179.99,0.00,-93.07)(7,0)
DEF POS Ppal=(455.41,-481.70,419.17,-174.93,0.87,-140.23)(7,0)
DEF POS PCOLOC=(418.00,479.40,221.80,-177.00,0.00,-90.00)(7,0)
```



Point 1

DEF POS Ppiram=(417.61,16.40,180.00,180.00,0.00,180.00)(7,0)

Point 2

DEF POS PINIC=(524.49,80.98,614.92,-179.99,0.00,-93.07)(7,0)

Point 3

DEF POS Ppal=(455.41,-481.70,419.17,-174.93,0.87,-140.23)(7,0)

Point 4

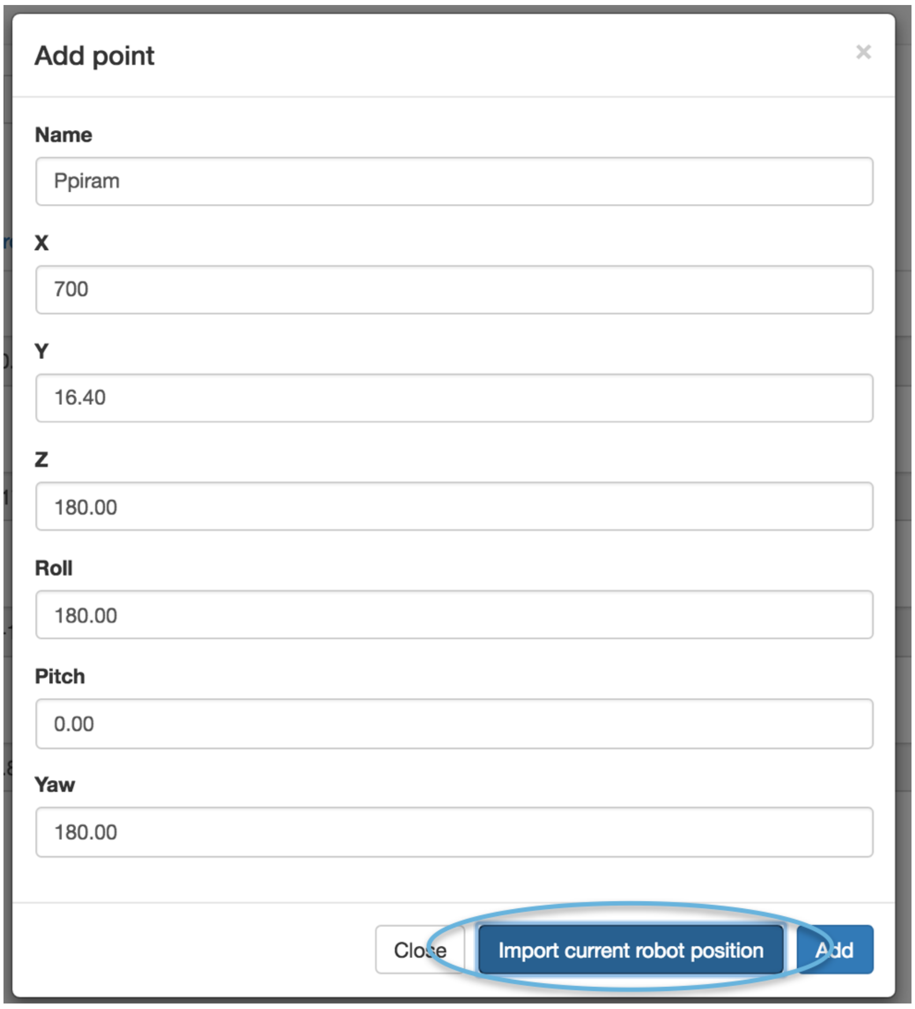
DEF POS PCOLOC=(418.00,479.40,221.80,-177.00,0.00,-90.00)(7,0)

Points file input

1.POS

Load points from .pos file.

4. We pressed the "Import current robot position" button from the points tab while the robot was in a known position. We validated that the position returned was correct.



Add point

Name

Ppiram

X

700

Y

16.40

Z

180.00

Roll

180.00

Pitch

0.00

Yaw

180.00

CHAPTER 8

Conclusions and Future work

The main goals of the project have been met as we have managed to develop a backend that can communicate successfully with the robot controller and instruct it to perform all the actions needed for robotics labs. We also created a working desktop frontend that was used to test the correct functioning of the backend. Finally, we developed a web application that makes full use of the backend and provides a usable interface to interact with the robot controller.

In this project we have employed and learned many key skills such as: low level device interactions using serial port protocols, developing modular robotics programs by leveraging ROS nodes and its publish subscribe mechanism, full stack web applications development, knowledge of compilers by creating a Melfa Basic IV parser, best programming practices and medium-large scale application development.

As for future work, the web application is fully functional, but it could use some more work to get feedback from the backend. For example it could show a spinning loading icon while the backend or robot is busy after the user performs an action. This way the application would not appear to be unresponsive when the action takes a while to execute.

Another improvement that could be made is to extend the database to store information about student groups and create another table where each group's programs and points are stored. This would let users save without having to physically save to disk or USB drive. It would also enable the application to save automatically to avoid losing changes if the web page is refreshed.

As a final improvement a new tab with a Mitsubishi RV-6S simulator could be added. This could be implemented with an HTML5 canvas and it would let students see a simulated execution of their code before running it on the robot, which would make it possible for them to do more work from home or while waiting for their turn to use the robot.

CHAPTER 9

APPENDIX

9.1 Full List of Commands Supported by the Backend

Those are the messages supported:

```
–––SINGLE INSTRUCTION–––
After receiving this message the backend expects to receive an instruction
in the following message. It will then command the robot to execute the received
instruction.

–––LOAD PROGRAM BEGIN–––
After receiving this message the backend expects to receive .MB4 code line
by line followed by a –––LOAD PROGRAM END––– when the whole program has been sent.

–––LOAD PROGRAM END–––
After receiving this message the backend stops waiting for more lines of the
program and loads it on the robot.

–––LOAD POINTS BEGIN–––
After receiving this message the backend expects to receive .POS point definitions
line by line followed by a –––LOAD POINTS END––– when the whole file has been sent.

–––LOAD POINTS END–––
After receiving this message the backend stops waiting for more point definitions
and loads the points on the robot.

–––RUN PROGRAM–––
After receiving this message the backend sends commands to run the code which is
currently loaded on the robot.
```

```
———DELETE———
After receiving this message the backend sends commands to delete the
loaded program and points on the robot.

// Movement commands
———MOV TOOL +X———
After receiving this message the backend moves the tool in the x direction
by a fixed increment.

———MOV TOOL −X———
After receiving this message the backend moves the tool in the negative
x direction by a fixed increment.

———MOV TOOL +Y———
After receiving this message the backend moves the tool in the y
direction by a fixed increment.

———MOV TOOL −Y———
After receiving this message the backend moves the tool in the negative
y direction by a fixed increment.

———MOV TOOL +Z———
After receiving this message the backend moves the tool in the z
direction by a fixed increment.

———MOV TOOL −Z———
After receiving this message the backend moves the tool in the negative
z direction by a fixed increment.

———REQUEST TOOL POSE———
After receiving this message the backend asks the robot for its current
tool pose. It then posts the tool pose in a ROS channel called "tool_pose"
with the following format "(x,y,z,roll,pitch,yaw)" so it can be retrieved
by the frontend.

———REQUEST JOINT STATE———
After receiving this message the backend asks the robot for it?s current
joint state. It then posts the joint state in a ROS channel called
"joint_state" with the following format "(j1,j2,j3,j4,j5,j6)" so it can
be retrieved by the frontend.
```

9.2 Pseudocode for Jog Operation with Multithreading

The following code implements multithreading for the jog operation and handling of multiple messages by using timestamps:

```
function jog_operation()
```

```
  while jog_operation_active
    if ts_last_finished_move < ts_move_x_pos
      move_x_pos()
      ts_last_finished_move = get_current_timestamp()
    else if ts_last_finished_move < ts_move_y_pos
      //The same for every possible movement
      ...
    end if
    sleep(0.1)
  end while
end jog_operation

function message_handler(msg)
  if move_joint_state_message(msg)
    if is_move_x_pos(msg)
      ts_move_x_pos = message_timestamp(msg)
    else if is_move_y_pos(msg)
      //The same for every possible movement message
      ...
    end if
    if not jog_operation_active
      jog_operation_active = True
      t = new Thread(jog_operation)
      t.start()
    end if
  else if move_joint_command(msg)
    //Tell jog operation to perform action
  else
    if jog_operation_active
      jog_operation_active = False
      t.join()
    end if
    //Handle message
  end if
end message_handler
```

9.3 Complete BNF Grammar Specification Melfa Basic IV

```
ID                  = {a–zA–Z}+
Real                = {0-9}+\.{0-9}+
Integer       = {0-9}+
String        = "{Letter}*"
Boolean       = TRUE
                    | FALSE
EOL                 ::= \n
```

```
                        | \r\n


<PROGRAM>       ::= <STMTS>

<STMTS>         ::= <STMT> EOL <LINES>
                        | <STMT>

<STMT>          ::= DEF DOUBLE <ID_LIST>
                        | DEF CHAR <ID_LIST>
                        | DEF FLOAT <ID_LIST>
                        | DEF INTE <ID_LIST>
                        | DEF POS <ID_LIST>
                        | SERVO ON
                        | SERVO OFF
                        | <REF> = <EXP>
                        | GOTO Label
                        | GOSUB Label
                        | Label
                        | RETURN
                        | IF <EXP> THEN <STMT>
                        | WHILE <EXP> EOL
                          <STMTS>
                          WEND
                        | FOR ID = <EXP> TO <EXP> [STEP <EXP>] EOL
                          <STMTS>
                          NEXT [<ID_LIST>]
                        | SELECT <EXP> EOL
                          {CASE <EXP> EOL <STMT> BREAK EOL}+
                          [DEFAULT EOL <STMT> BREAK EOL]
                          END SELECT
                        | MOV ID [, [+-] Integer]
                        | MVS ID [, [+-] Integer]
                        | END
                        | DLY Integer
                        | HALT
                        | HOPEN Integer
                        | HCLOSE Integer
                        | JOVRD Integer
                        | OVRD Integer
                        | SPD Integer

<ID_LIST>       ::= <ID>, <ID_LIST>
                        | <ID>

<EXP>           ::= <AND_EXP> OR <EXP>
                        | <AND_EXP>
```

```
<AND_EXP>       ::= <NOT_EXP> AND <AND_EXP>
                  | <NOT_EXP>

<NOT_EXP>       ::= NOT <COMP_EXP>
                  | <COMP_EXP>

<COMP_EXP>      ::= <ADD_EXP> = <COMP_EXP>
                  | <ADD_EXP> < <COMP_EXP>
                  | <ADD_EXP> <= <COMP_EXP>
                  | <ADD_EXP> > <COMP_EXP>
                  | <ADD_EXP> >= <COMP_EXP>
                  | <ADD_EXP> <> <COMP_EXP>
                  | <ADD_EXP> >< <COMP_EXP>
                  | <ADD_EXP>

<ADD_EXP>       ::= <MULT_EXP> + <ADD_EXP>
                  | <MULT_EXP> - <ADD_EXP>
                  | <MULT_EXP>

<MULT_EXP>      ::= <NEG_EXP> * <MULT_EXP>
                  | <NEG_EXP> / <MULT_EXP>
                  | <NEG_EXP> MOD <MULT_EXP>
                  | <NEG_EXP>

<NEG_EXP>       ::= - <POWER_EXP>
                  | <POWER_EXP>

<POWER_EXP>     ::= <POWER_EXP> ^ <VAL>
                  | <VAL>

<VAL>           ::= ( <EXP> )
                  | <REF>
                  | <CONST>
                  | sin ( <EXP> )
                  | cos ( <EXP> )
                  | tan ( <EXP> )
<CONST>         ::= Integer
                  | String
                  | Real

<REF>           ::= ID.x
                  | ID.y
                  | ID.z
                  | ID
```

9.4 Typo Suggestion Pseudocode

```
function levenshtein_match(tk)
  //possible_keywords is a list of keywords that are valid at the
  //start of a statement
  best_match = None
  best_match_distance = INFINITY
  for keyword in possible_keywords
    if levenshtein_distance(tk, keyword) < MIN_DISTANCE
       and levenshtein_distance(tk, keyword) < best_match_distance
      best_match = keyword
      best_match_distance = levenshtein_distance(tk, keyword)
    end if
  end for
  return best_match
end levenshtein_match

function A_ASSIG()
  tk = next_token()
  if tk == '='
    //Further error checking
    ...
  else
    lm = levenshtein_match(tk)
    if lm
      error("Invalid statement. Did you mean " + lm +
            " instead of " + tk + "?");
    else
      error("Expected assignment with =")
    end if
  end if
end A_ASSIG
```

Bibliography

[1] T. Author, "22424. robotica (2016-17)," 2016, [Online; accessed 23-July-2016]. [Online]. Available: http://estudis.uib.cat/grau/informatica/GEIN-P/22424/index.html 2

[2] E. W. Weisstein, "Euler angles," 20016, [Online; accessed 26-July-2016]. [Online]. Available: http://mathworld.wolfram.com/EulerAngles.html 1

[3] "Robot operating system," 2016, [Online; accessed 23-July-2016]. [Online]. Available: http://www.ros.org 4.2.1

[4] Stwirth and M. Massot, "Arm control repository," 2016, [Online; accessed 23-July-2016]. [Online]. Available: https://github.com/srv/arm_control/tree/fuerte/src 4.3

[5] "Rosbridge suite," 2016, [Online; accessed 24-July-2016]. [Online]. Available: http://wiki.ros.org/rosbridge_suite 6.1

[6] "Roslibjs," 2016, [Online; accessed 24-July-2016]. [Online]. Available: http://wiki.ros.org/roslibjs 6.1

[7] "Basic bnf grammar," 2016, [Online; accessed 24-July-2016]. [Online]. Available: https://rosettacode.org/wiki/BNF_Grammar 6.6.2

[8] J. G. Sastre, "Melfa basic iv manual," 2016, [Online; accessed 24-July-2016]. [Online]. Available: http://dmi.uib.es/~jguerrero/instMelfa.pdf 6.6.2

[9] G. Navarro, "A guided tour to approximate string matching," 2001, [Online; accessed 24-July-2016]. [Online]. Available: http://repositorio.uchile.cl/bitstream/handle/2250/126168/Navarro_Gonzalo_Guided_tour.pdf 6.6.4