Master's thesis

Johanne Døvle Kalland

The NTNU Cyborg 4.0

The Behaviour Module & Behaviour Trees

Master's thesis in Cybernetics and Robotics
Supervisor: Associate Professor Sverre Hendseth,
PhD Candidate Martinius Knudsen
June 2020

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Johanne Døvle Kalland

The NTNU Cyborg 4.0

The Behaviour Module & Behaviour Trees

Master's thesis in Cybernetics and Robotics
Supervisor: Associate Professor Sverre Hendseth,
PhD Candidate Martinius Knudsen
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Task Description

The main goal of all the work done with the Cyborg, whether that is developing a new module, creating new hardware or improving parts of the previous system, is to create a better and more advanced Cyborg than before. The work in this thesis focuses on the behaviour of the Cyborg and its LED-dome and audio module, and is combined with the work done by fellow students to bring the Cyborg closer to a state where it is ready for demonstrations and to act as a mascot for NTNU.

The student shall:

- Reevaluate the Cyborg as it was at the start of this thesis.
- Look at how a system can be implemented on the Cyborg using behaviour trees, evaluate different behaviour tree libraries, and compare them to the existing previous use of behaviour trees on the Cyborg.
- Investigate how the use of psychology and colour can make the Cyborg more attract more attention, and how it affects people's emotions.
- Implement PAD values for each of the behavioural presets in the behaviour launch file.
- Create a complete setup-file and a requirements file for the Cyborg software system.
- Expand the Cyborgs library of behavioural presets and create new animations for the LED-dome and add new audio files to the Audio node.

Preface

This thesis is written at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology for the NTNU Cyborg project in the course TTK4900 - Engineering Cybernetics, Master's Thesis. The Cyborg project is developing a robot connected to live neural cells grown at St. Olav's Hospital in Trondheim. The purpose of this work has been to continue the development of the cyborg with special attention to the behaviour module. The work in this thesis is a continuation of the author's specialisation project from the autumn of 2019 [23].

I would like to thank Sverre Hendseth for his guidance in writing this report, Martinius Knudsen for his support regarding all aspects of the Cyborg, and Casper Nilsen, Lasse Göncz and Ole Martin Brokstad for being great coworkers on the project and people to bounce ideas off of.

The Covid-19 situation occurred in the middle of the work with this thesis and created a unique situation, not just for everyone working on their master's theses, but for our society as a whole. This meant that physical access to the Cyborg was no longer possible as NTNU was mostly shut down and students and employees ordered to shelter-at-home. Because of this, some parts of the work took a different direction than previously planned.

Abstract

The NTNU Cyborg project aims to create a cyborg by enabling communication between a robot and living nerve cells. This cyborg will then become a mascot for both the Norwegian University of Science and Technology and the departments involved. This thesis adds incremental improvements to the robotics part of the project, while also investigating the future possibilities of the Cyborg.

Since the project's creation, many students have created and implemented numerous parts, both hardware and software. Because of this, many of those older implementations use libraries that have reached their end of life. The amount of work needed to bring the Cyborg up to date with the latest releases of those dependencies and for it to use Ubuntu 20.04 Focal Fossa and ROS Noetic Ninjemys has been evaluated. A file listing all the Cyborg's dependencies and software has been created. This file doubles as documentation, listing all libraries and their versions needed, and as a setup script for installing the Cyborg on a new system. In studying the possibilities of upgrading the software and collecting all libraries, a reevaluation of the Cyborg as it was at the onset of this thesis was done. As this thesis studies the use of behaviour trees on the Cyborg, the behaviour module has been examined. Because the behaviour trees are not ready to replace the current module, the list of behavioural presets has been expanded and they have had PAD values added. Different behaviour tree libraries were investigated and they were compared to the library used on the Cyborg previously. The usage of colour and how it impacts people's perception of the Cyborg and emotions they can trigger was studied. This lays a groundwork and works as a reference work for future application of colour in implementing behaviours and interpreters for the LED-dome.

Sammendrag

Prosjektet NTNU Cyborg has som mål å muliggjøre kommunikasjon mellom en robot og levende nerveceller. Cyborgen vil da fungere som en maskot for NTNU og instituttene som deltar i prosjektet. Denne oppgaven bidrar med inkrementelle forbedringer til robotikkdelen av prosjektet i tillegg til å undersøke fremtidige løsninger for Cyborgen.

Mange studenter har jobbet på prosjektet siden opprettelsen av prosjektet, og har implementert utallige deler, både programvare og maskinvare. På grunn av dette bruker mange av de tidligere delene biblioteker som ikke lengre blir videreutviklet. For å få Cyborgen over på nyere programvare, da spesielt Ubuntu 20.04 Focal Fossa og ROS Noetic Ninjemys, kreves det mye arbeid og dette har blitt evaluert i tillegg til om det er verdt å oppdatere. En fil som inneholder alle biblioteker som brukes har blitt laget og den fungerer som dokumentasjon på biblioteker og hvilke versjoner som brukes. Videre fungerer den som et skript som kan kjøres for å installere alt som trengs for å kjøre Cyborgen på en ny pc. På begynnelsen av arbeidet ble det tatt en evaluering av Cyborgen for å gjøre rede på hva det var ønskelig at det ble jobbet med, eventuelt om noe skulle fjernes. Denne oppgaven undersøker også bruken av adferdstrær på Cyborgen og om de kan erstatte oppførselsmodulen som er i bruk nå. En enkel implementasjon av et tre ble laget, men det er ikke klart til å fullstendig erstatte den nåværende modulen. Modulen bruker en liste med forhåndsbestemte oppførsler, men bare en håndfull av disse hadde PAD verdier å sende som tilbakemelding til følelses modulen etter at de var utført. Alle har nå fått PAD verdier, med unntak av noen få hvor de ikke er nødvendige. Bruken av farger og hvordan de påvirker folks oppfatning av Cyborgen og hvilke følelser de kan utløse ble undersøkt. Denne undersøkelsen kan fungere som et oppslagsverk for fremtidige anvendelse av farger i implementasjoner av oppførsel og animasjoner for LED-hodet til Cyborgen.

Contents

Chapter 1

Introduction

The NTNU Cyborg is a joint project between the Department of Engineering Cybernetics, the Department of Neuromedicine and Movement Science, and the Department of Computer Science at the Norwegian University of Science and Technology. Its goal is to develop a cyborg which uses living nerve tissue which communicates with a robot, which is to be used as a research platform and a mascot for both NTNU and the participating departments. The development of the project is done by PhD and MSc students and supervised by NTNU researchers and professors. The nerve tissue is grown on a Micro-Electrode Array (MEA) at St. Olav's Hospital and previous students have created a two-way communication channel over the internet between them and the robot which is located at NTNU campus Gløshaugen. The MEA enables us to read the tiny changes in voltage created by the cells. It can also stimulate the cells giving the possibility for a feedback-loop with the robot. The project utilises ROS (Robotic Operating System) as its base for the software where packages are modular and can be removed or added with relative ease. The robot is built on a base called Pioneer LX from MobileRobots, with a shell created for the Cyborg added on top of the base. While there is a distinction between the names *robot* and *Cyborg*, they are often used interchangeably, and because it does not have an official name it is mostly called the Cyborg.

The work done this spring is aimed at continuing our own work, while at the same time building on other students work from previous years. Here, the main goal has been to create a smoother experience of the Cyborg as well as updating existing and adding new software, and through this bring the project closer to its final state.

Presented in this report are the implementations of new LED-dome interpreters and audio files, and propositions for different behaviours they can be used in. The possibilities for upgrading ROS and Ubuntu and what impact that could have on the Cyborg are investigated. The libraries and dependencies used on the Cyborg are collected into a requirements file and a setup script which can be run to install the Cyborg on a new system. Different behaviour tree libraries are studied and compared, and a proposition for how a behaviour tree can be implemented are presented. A study of the psychology of colours and their effect of people's emotions will be presented and the results can as a reference work for future application of colour in the Cyborgs LED-dome interpreters.

1.1 Ongoing Work

Described here is the work that has been done in parallel to this thesis by other students working on the NTNU Cyborg project.

The Navigation Stack

The work done with the navigation stack has been to continue the re-implementation of the navigation system and optimising the localisation performance. Because the navigation stack was using proprietary code, it is now based on the ROS navigation stack using ROS nodes that together perform path planning, obstacle avoidance, localisation and mapping. [17]

Object Detection

A computer vision module designed for the Cyborg has been implemented on a Jetson TX1 development board and uses a ZED stereo camera for a live video feed. The module is designed to detect humans to pave the way for the Cyborg to become a socially intelligent robot. Objects detected by the module can be used in the controller and behaviour modules to trigger specific behaviours, for example, interact with people. It can also be used when navigating and provide the Cyborg with an additional form of obstacle detection. [7]

GUI Module

The goal of the work with the website was to support remote monitoring and control of the Cyborg, as well as increasing the level of automation of the robot. The website is implemented as a cloud-based, reactive single-page application. It allows the Cyborg to be manoeuvred remotely and in real-time with an interactive map using either keys or the on-screen joystick. The GUI also allows for changing the state in the behaviour system, SMACH state machine and changing the mode of operation. [34]

1.2 Previous Work with the Cyborg

This section is based on a similar section from the author's specialisation project. Changes such as the work done during the three specialisation projects in the autumn of 2019 have been added.

There have been several students working on the cyborg during the last years. All have been working on it as part of their master's thesis and/or specialisation project. Last year the cyborg was stripped of all unnecessary components to make a base on which the cyborg can be further developed. Other previous work have involved a selfie stick, an iris, arms for the cyborg, face recognition, an artificial muscle and an Xbox Kinect. A quick explanation of the previous years' work is presented below.

Autumn 2019 - Because major parts of the navigation system on the Cyborg consisted of proprietary code, an effort to develop a new navigation stack was begun, and show to be working in the simulator. A proof-of-concept website was created as a replacement for the Mode Selector box, in addition to creating a remote GUI interface for the Cyborg.

Spring 2019 - Collected all non-redundant parts of the cyborg into a working base for further development. The motivation was a simpler software structure. Old modules were improved, some kept as they were, and new modules were developed and added such as the behaviour module [5][4].

Spring 2018 - Contributed with incremental improvements to the robotics part of the cyborg. The now unused controller system based on behaviour trees was developed. The goal was to represent a more realistic behaviour than the existing system at that time. A program to visualise the behaviour tree while running was also developed. Lastly, a system for object detection and a classification system using neural networks was added [30].

Spring 2017 - A controller for the robot was made, as earlier modules were able to simultaneously run different modules. This controller is the basis for the current controller and is a state machine controller. It used a PAD emotion state model to decide the mood and what to other modules to run. A navigation module using ROSARNL was also implemented [3]. Work was also done to ready the cyborg for presentations. This became the Presentable Robot, and at the same time introduced the idea of a core on which students could implement their projects. This is where the, at that time called Startup Box now Mode Selector Box, was made [51].

Spring 2016 - Work on implementing speech recognition was done this year. This to have the robot communicate with the public. Research was done to examine what the best option for a speech recognition tool would be, and also to convert speech to text. The result was it being able to hold a conversation with informative and humorous answers [25].

1.3 Work Not Mentioned Any Further

As the Cyborg is a project worked on by many students over several years there will always be some work that is done to combine those projects, which is not necessarily part of the main work. Some of this work includes fixing minor and major errors that might occur, merging different work branches into each other and improving upon minor details from previous work. The work in this thesis has been no exception. When integrating the work done with the website and the new navigation stack reached points where they could be added to the other software, work was needed to change the existing software which used or depended on the new parts so that they would work properly together.

Chapter 2

Background

This background chapter is based on the same chapter from the author's specialisation project from the autumn of 2019 [23]. As most of the hardware and software used on the Cyborg have not changed. New sections covering new software and hardware have been added.

2.1 Hardware

Here, the parts of the hardware of the cyborg which have been relevant for the work in this thesis are presented.

2.1.1 The Cyborg Base - Pioneer LX

The base of the Cyborg is a mobile robot platform called Pioneer LX from MobileRobots [32][31]. It is a general-purpose indoor platform designed to be able to work around people and is based on the Adept Lynx AIV, also from MobileRobots. It serves as the moving and navigation platform and main computer for the Cyborg on which everything else is mounted. Some software libraries are included which are mentioned further in section 2.2.3. The robots are

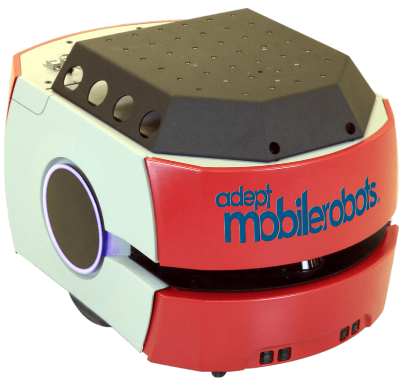


Figure 2.1: The Pioneer LX robot. Image courtesy of [41].

made for easy integration of custom accessories and sensors, and is programmable.
Some of the reasons it was chosen were because it is able to carry up to 60 kg and the on-board computer can run both Windows and Linux. It also features different sensors such as a laser to map its surroundings. Other features are:

- Intel D252 64-bit dual core 1.8GHz CPU and an integrated graphics processing unit.
- 2 GB DDR3 RAM
- Wireless Ethernet connection.
- Front- and rear-facing sonar sensors, and front bumper panel.
- SICK S300 laser scanner.
- Several USB 2.0 ports, a VGA monitor port, 16 In/16 Out digital I/O, and 4 In/4 Out analogue I/O.
- 60 Ah battery, capable of powering the robot for 13 hours.
- Charging and docking station which allows the robot to charge itself.
- Joystick for manual steering and control.

2.1.2 Led-Dome

The led-dome is an essential part of the Cyborg. Without it, the cyborg would be just another robot driving around and would not be as striking as it is now. In fig. 2.2 the led-dome can be seen on top of the robot body. The dome is made up of two plastic shells, the inner and the outer. On the outside of the inner shell, strips of LEDs of the type WS2812B have been glued in a zig-zag pattern amounting to $791 \cdot 3$ RGB LEDs. Each of them is individually addressable. The byte-array sent to the led-controller from the domecontrol ROS node can be accessed the same way as any other array in Python. The outer shell is made of VIVAK PETG and was vacuum formed along with the inner shell and then frosted using sandpaper. The process of making the led-dome can be found in [1]. The dome is controlled by a 5V PWM-signal and requires an external power supply of 5V as the led-controller is not able to produce sufficient power.

It is important to differentiate between the led-dome, the led-controller and the ROS node controlling the led-dome, as it is easy to be confused. In this report *led-dome* will be used for the hardware that is the dome, *led-controller* will be used for the hardware that is controlling the LED strips, and *controller node* will be used for the ROS node. Be aware that *led-controller* might also be used for the software on the led-controller as it is not part of the ROS node.

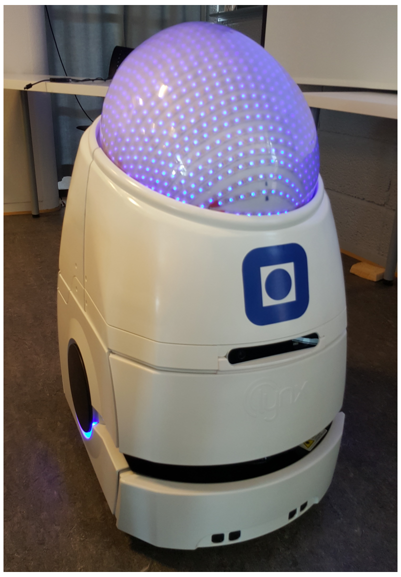


Figure 2.2: The Cyborg with the led-dome.

2.1.3 Mode Selector Box

The Mode Selector box started as a start-up box made by Jørgen Waløen for his master's thesis [51] that ran the scripts corresponding to the choices at start-up. These choices were ARIA Demo by MobileRobots and launch of the ROS nodes. He wrote a library in C for the Arduino Nano and OLED screen, compiled it with a Makefile and uploaded it with avrdude. Last spring the box was updated so it would be able to switch between modes without the need to shut down the whole robot in between. Another mode, called `ARNL`, was also added. As this change added functionality outside the scope of its original name, the box was renamed Mode Selector box. A description of all changes made in 2019 can be found in [5].

2.1.4 MEA2100 - System Microelectrode Array

The MEA2100-system is made by Multichannel Systems and is a versatile *in vitro* system made for extracellular recording from microelectrode arrays

[47]. It can record cardiac or neuronal cultures, stem cells, or cardiac or brain slices. MEA technology is a powerful tool in electrophysiology research and is based on an idea from the 1970s. The system gives real-time feedback from 60-channels with a 24-bit resolution. Every electrode in the MEA2100-systems is bidirectional, meaning that it is possible to stimulate the cells on each electrode. Neural cells are grown in the MEA system at the neuroscience department at St. Olav's and provides the data used for visualisation on the led-dome. The system interprets and sends the signals through a computer, which then sends it to the cyborg over a server.

2.1.5 NodeMCU ESP-32S

The NodeMCU ESP-32S is a development board made by NodeMCU [35] based on the ESP-WROOM-32 module by Espessif Systems [46]. The board has a 32-bit double-core architecture with a clock speed of 240 MHz. It has a USB Micro-B connection used for writing to and reading from the board in addition to power. It operates with 3.3V and has 38 I/O pins which support a variety of uses and protocols such as UART, PWM, and output from the on-board DA-converter. Earlier, instead of the NodeMCU ESP-32, the led-controller was an Arduino Mega 2560 which was replaced in the spring of 2019.

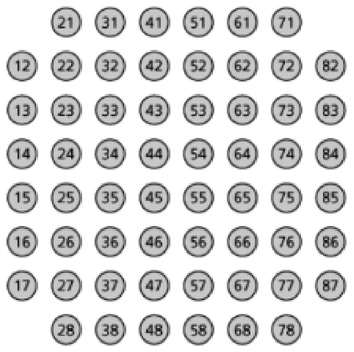


Figure 2.3: Standard electrode numbering of the 60-electrode MEA.

Directly from NodeMCU it runs Lua TOS real-time system and is programmed in Lua using the Lua IDE. Lua RTOS is designed with three-layers with a Lua interpreter on top, a real-time micro-kernel in the middle, and at the bottom a hardware abstraction layer. The middle layer is powered by FreeRTOS. Installing a minor add-on for the Arduino IDE, allows for programming of the EPS-32 in Arduino IDE with the Arduino programming language [21], changing out the top layer. This allows for much easier usage of the ESP-32 as there is no need to learn another programming language.

The Arduino code is only run on one core. Because the ESP-32 has two cores, programming must be done using FreeRTOS tasks and methods, otherwise only one core will be utilized.

Together with the ESP-32, the LED-controller consists of a small circuit with a CMOS-buffer. This is because the LED strips require a 5V PWM-signal to operate while the ESP-32 only provides a 3.3V signal. Therefore, a level conversion is needed. The reason a buffer is used and not a passive converter with a pull-up resistor, is because a pull-up resistor is to slow to handle the high operational frequency of the LEDs. The type of buffer used is a quad bus buffer of the type *74VHCT125AT*. A detailed description of how the software of the LED-controller works can be found in [4].

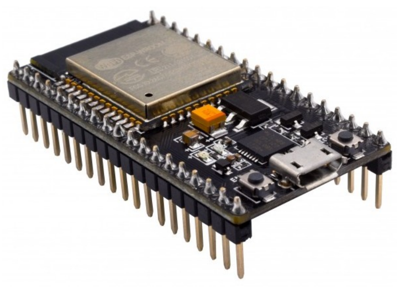


Figure 2.4: NodeMCU ESP-32S. Image courtesy of [36].

2.2 Software

All code for the cyborg can be found on the NTNU Cyborg's GitHub page [49]. There each ROS node contains a `README` file explaining the basic details of the node. In addition, more in-depth documentation can be found in the wiki.

2.2.1 FreeRTOS

FreeRTOS is a real-time operating system made for use in embedded systems. The idea was to design an os which would need very little space in the memory, allowing for fast execution. It is distributed for free under the MIT License [27]. FreeRTOS contains methods for parallel threads, tasks, timers, mutexes and semaphores. Threads may also be prioritised. For the cyborg, a small install of the NodMCU ESP-32 is needed in the Arduino IDE to program with FreeRTOS. This because the ESP-32 is already running FreeRTOS in the middle layer, as mentioned in section 2.1.5. To give a quick introduction to the methods used on the led-controller some examples are given and explained. A further description of FreeRTOS' methods can be found in [15].

```c
xTaskHandle xHandle;
static unsigned char ucParameter;

xTaskCreate(
    vTaskLoop,        //Loop-funksjon
    "TaskNavn",       //Task navn
    STACK_SIZE,       //Stack str
    &ucParameter,     //Peker brukt som parameter til task
    PRIORITY,         //Priotiteten
    &xHandle,         //Task handle som det returneres til
    0);               //Core running the task

void vTaskLoop(void * parameter){
    for ( ;; ){
        /* Task kode her */
    }
}
vTaskDelete( xHandle );
```

- The first line sees a task created. A task handle is like a reference for the task to be used to delete or create it.
- `vTaskLoop` is the main loop of the task, and works like the *loop()*-function on Arduinos. It will run as long as the associated task exists.
- `vTaskDelete` takes the handle of a task and deletes it.
- `xTaskCreate` creates the task. If one or several of the parameters taken by the function is not needed, they can be sent as `NULL`.
- The parameter, here declared as *ucParameter*, must exist for the entire lifetime of the task. In this case, it is then declared as a static.

FreeRTOS contains several conventions for naming constants, functions, and more. The two most used are the prefixes of the functions `xTaskCreate` and `vTaskLoop` in the examples above. The prefix v means that the return type of the function is `void`. x denotes some result, which is often a handle for a task or a queue [8].

2.2.2 Arduino IDE

The Arduino IDE (integrated development environment) supports coding of a massive number of development boards, not only made by Arduino, in their own language which is a mix of C and C++. It provides a large number of built-in libraries, as well as any modified libraries which have to be open-source. With the installed add-on, the ESP-32 can be programmed with the IDE using C/C++ and FreeRTOS.

2.2.3 Software with the Pioneer LX Base

The Pioneer LX robot comes with several pre-installed and useful tools, some of whom are mentioned here. ARIA is open-source, while some are free, but most must be purchased. Since MobileEyes was discontinued one of the biggest issues have been the accessibility of these tools. Changing this is the work described in [18] and [17].

- **ARNL** - *Advanced Robot Navigation and Localisation* is built on top of ARIA, made by MobileRobots. It is a set of software packages and used for localisation, by keeping track of where the robots is, and navigation, allowing the robot to receive a give specific destination. This is now being replaced by the new navigation stack

- **ARIA** - *Advanced Robot Interface for Applications* can dynamically control a robot's heading, velocity, and other motion parameters. In addition, it can receive operating data sent by the platform. All this is done through its high-level infrastructure or low-level commands.

- **MobileSim** - Simulator for the Pioneer LX base.

- **Mapper3** - Maps needed in ARNL can be edited and converted to the right format with this tool.

- **MobileEyes** - Provides a GUI for the robot base. It can remotely control and monitor the Pioneer LX, in addition to create maps. The program can be run from a computer and connect to the robot via the network.

2.3 Cyborg Software Structure

The cyborg consists of many modules, which may be hard to get a grasp of the first time reading them. Here is therefore a quick explanation of each node. In fig. 2.5 is a simplified class diagram of the whole cyborg, with subscribed to and published topics. Figure 2.6 shows the relation between ROS nodes and topics. Both can also be found on the NTNU wiki [48]. Since the onset of this thesis some of these have been added, changed, or removed as the software has been changed. They have been included here to provide insight into previous iterations and continuity of the Cyborg software.

Audio - A ROS node in charge of playing audio files, and also contains text-to-speech functionality.

Behaviour - A ROS node that adds and executes behavioural presets. It exploits the emotional state of the cyborg and provides emotional feedback. New behavioural presets and configurations are added in the `behavior.launch` file. Also decides what future states to execute.

Event Scheduler - A ROS node which provides functionality related to publishing scheduled events. It also monitors other system events like low battery.

Primary States - Gathers action server states too complex for the behaviour module, and states that do not produce outputs. Can execute state changes, and provides emotional feedback to the controller. A ROS node.

Command - A ROS node, implemented, but not in active use at the moment. It is a command module run on an external computer to monitor the cyborg. It is not in use while the system is running, but can be used as a tool when testing.

Controller - The main state machine on the cyborg which manages all other nodes. Organises all actions in the state machine, handles the emotional system and motivator. Also a ROS node.

Navigation - A ROS node which handles all high-level navigational execution on the cyborg through the action server. The node `cyborg_navigation` is the navigation controller implemented by students working on the Cyborg, while `navigation` is a collection of open-source packages used by the navigation node.

Mode Selector - Not a ROS node, but contains the code used on the external box used to choose which mode the cyborg should run. Removed in 2020.

LED Dome - Controls the behaviour of the led-dome and sends output to the led-controller to set the led strips. It does not choose the different visualisation modes, but receives them on the topic `/cyborg_visual/domecontrol` from the behaviour module.

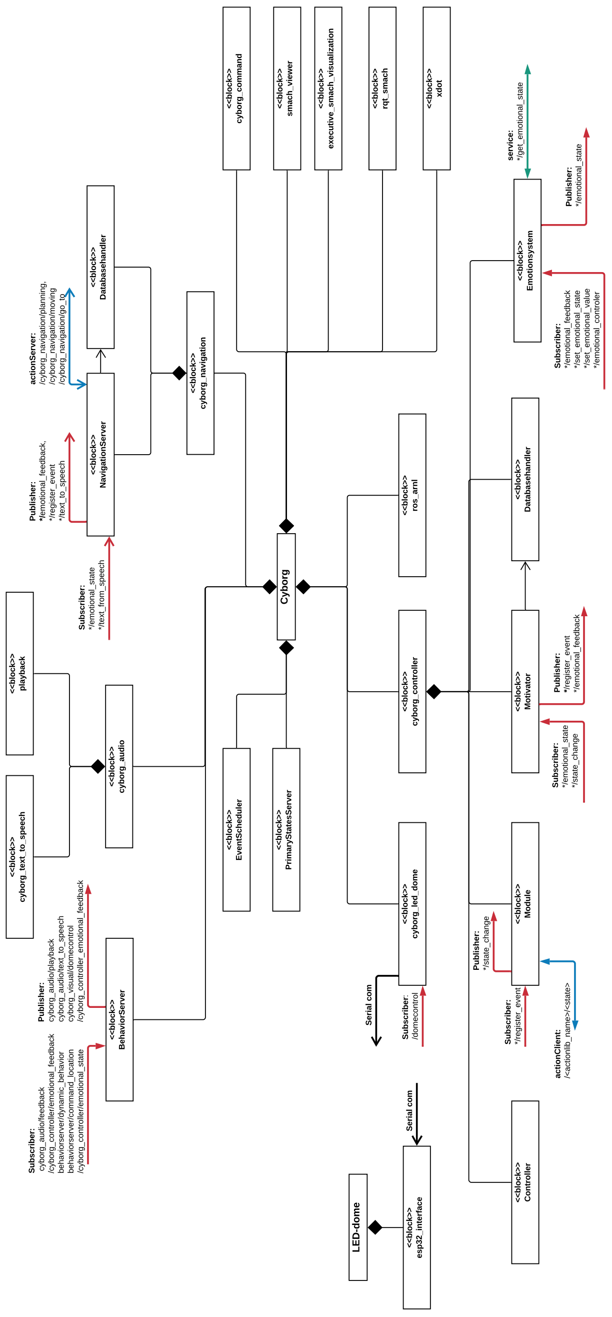
ROS Arnl - A ROS node developed by MobileRobots which handles communication between the robots sensors, actuators and navigation library, and the rest of the ROS nodes. Because of the new navigation stack, it is being replaced by open-source nodes which provide the same functionality, and should not be needed any longer.

ROSARIA - A ROS node and provides a ROS interface for most MobileRobots bases such as the Pioneer LX.

Commander - The Cyborg's node for communicating with the new website, created in 2020. It also acts as a replacement for the `Mode Selector` together with the website.

2.4 ROS - Robotic Operating System

ROS is an open-source robotics middleware. It provides services designed like those of an operating system, though it is in itself not an operating system. It is a flexible framework for working with and developing robot software [40]. ROS works well with modular systems as each node can be removed, added or edited without affecting the others, and for control of distributed systems. Nodes of the same robot can be run on different hardware and written in different languages as ROS is language- and platform-independent. All this allows for easy scalability. Their main goal is to provide support and easy reuse of code for research and commercial use.



<<block>> cyborg_command
<<block>> smach_viewer
<<block>> executive_smach_visualization
<<block>> rqt_smach
<<block>> xdot
actionServer: /cyborg_navigation/planning, /cyborg_navigation/moving /cyborg_navigation/go_to
<<block>> Databasehandler
Publisher: */emotional_feedback, */register_event */text_to_speech
<<block>> NavigationServer
<<block>> cyborg_navigation
Subscriber: */emotional_state */text_from_speech
service: */get_emotional_state
Publisher: */emotional_state
<<block>> Emotionsystem
Subscriber: */emotional_feedback */set_emotional_state */set_emotional_value */emotional_controler
<<block>> Databasehandler
<<block>> ros_arnl
Cyborg
<<block>> playback
<<block>> cyborg_text_to_speech
<<block>> cyborg_audio
<<block>> EventScheduler
<<block>> PrimaryStatesServer
<<block>> cyborg_controller
<<block>> Motivator
Publisher: */register_event */emotional_feedback
Subscriber: */emotional_state */state_change
Publisher: cyborg_audio/playback cyborg_audio/text_to_speech cyborg_visual/domecontrol /cyborg_controller_emotional_feedback
Subscriber: cyborg_audio/feedback /cyborg_controller/emotional_feedback behaviorserver/dynamic_behavior behaviorserver/command_location /cyborg_controller/emotional_state
<<block>> BehaviorServer
<<block>> cyborg_led_dome
Serial com
Subscriber: /domecontrol
<<block>> Module
Publisher: */state_change
Subscriber: */register_event
actionClient: /<actionlib_name>/<state>
Serial com
LED-dome
<<block>> esp32_interface
<<block>> Controller

Figure 2.5: A simplified class diagram of the whole cyborg. From the NTNU wiki [48].

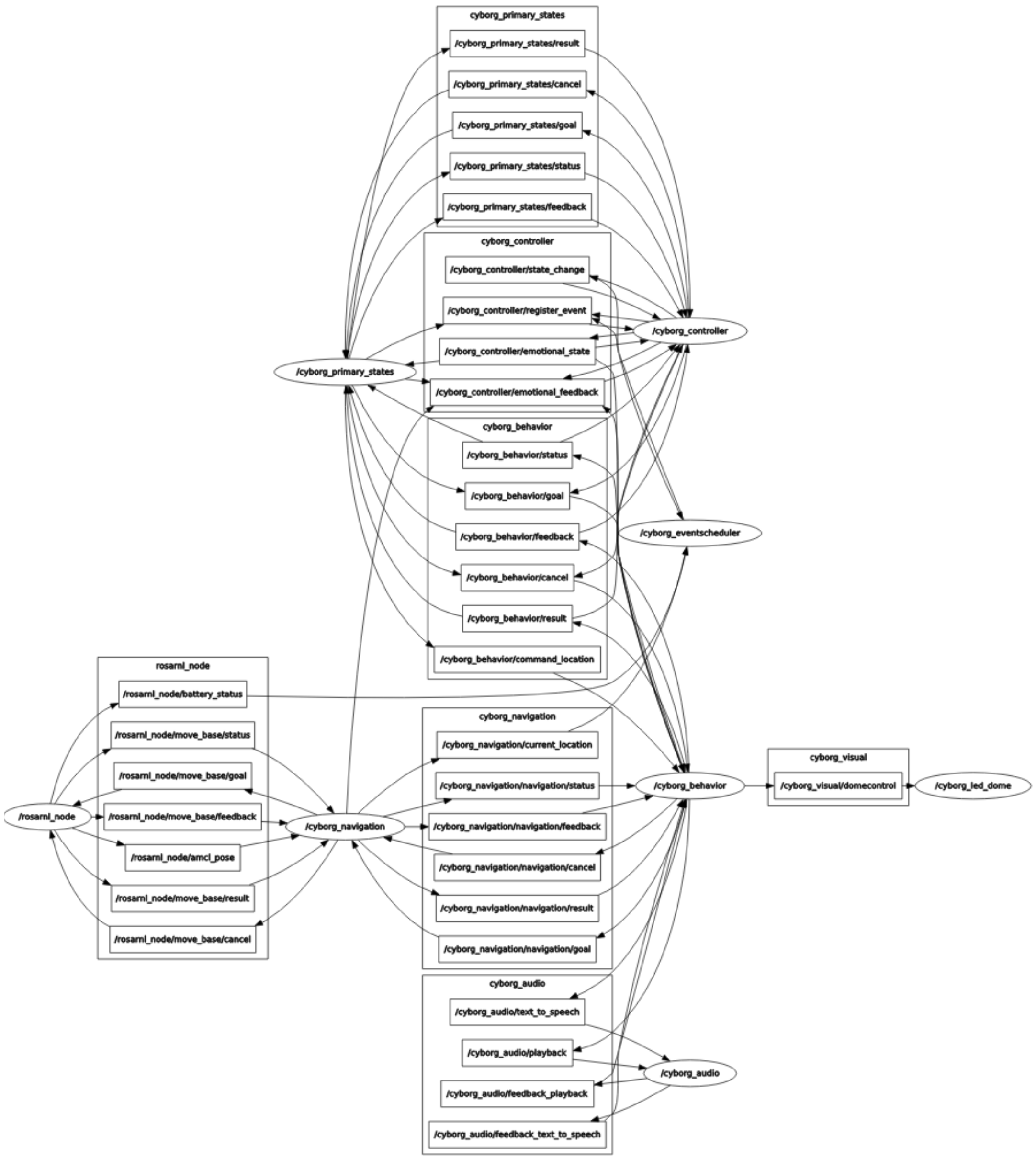

cyborg_primary_states
/cyborg_primary_states/result
/cyborg_primary_states/cancel
/cyborg_primary_states/goal
/cyborg_primary_states/status
/cyborg_primary_states/feedback
cyborg_controller
/cyborg_controller/state_change
/cyborg_controller/register_event
/cyborg_controller
/cyborg_controller/emotional_state
/cyborg_primary_states
/cyborg_controller/emotional_feedback
cyborg_behavior
/cyborg_behavior/status
/cyborg_behavior/goal
/cyborg_behavior/feedback
/cyborg_eventscheduler
/cyborg_behavior/cancel
/cyborg_behavior/result
/cyborg_behavior/command_location
rosarnl_node
/rosarnl_node/battery_status
/rosarnl_node/move_base/status
/rosarnl_node/move_base/goal
/rosarnl_node
/rosarnl_node/move_base/feedback
/rosarnl_node/amcl_pose
/rosarnl_node/move_base/result
/rosarnl_node/move_base/cancel
/cyborg_navigation
cyborg_navigation
/cyborg_navigation/current_location
/cyborg_navigation/navigation/status
/cyborg_navigation/navigation/feedback
/cyborg_navigation/navigation/cancel
/cyborg_navigation/navigation/result
/cyborg_navigation/navigation/goal
/cyborg_behavior
cyborg_visual
/cyborg_visual/domecontrol
/cyborg_led_dome
cyborg_audio
/cyborg_audio/text_to_speech
/cyborg_audio/playback
/cyborg_audio
/cyborg_audio/feedback_playback
/cyborg_audio/feedback_text_to_speech

Figure 2.6: Overview of the different ROS topics and nodes on the cyborg. Oval modules are nodes and the rectangles are topics. Outgoing arrows from the nodes are publishers and incoming are subscribers.

Since its initial release in 2007, the number of packages ready for download has been steadily growing. There are now two different versions ROS and ROS2. ROS releases one new release every two years, while ROS2 does so every six months. The lack of real-time support in the first version of ROS has been addressed in ROS2, which also has support for Windows. There is very little which is actually core ROS, beyond the general structure. Users can configure tools and libraries to fit their application area and robot.

2.4.1 The Concepts of ROS

ROS contains many different methods and concepts. Many of which are used on the cyborg. The most important concepts are presented here.

ROS Master

The ROS Master is an integral part of any ROS system. It can be run with the `roscore` or `roslaunch` commands, which loads the Master along with the other essential parts. The Master provides registration and naming services, as well as keeping track of subscribers and publishers. Its role is to make all information needed by the nodes available. The parameter server is also provided by the Master, but it will not be described in detail here.

Nodes

Nodes are the processes of ROS that perform any computation, combined by the communication streams services, topics or the parameter service to form a graph. They can be compared to atoms connected together to form molecules, and are the part that makes ROS so modular. They can easily be restarted in the case of a crash and can be switched out just as easily. Nodes communicate peer-to-peer through the communication streams and can identify each other by name the same way a Unix system uses a file path to locate a file. ROS nodes are part of a package. Most often each package contains only one node, usually with the same name as the package, but that is not a must. A package can contain as many nodes as needed.

Topics

Topics are the channels nodes use to send messages over to communicate. Topics are defined by name, just like messages. They are asynchronous and follow a publish-subscribe model, where there is no limit to the number of topics a node can subscribe and publish to. A node can publish on a topic it is also subscribing to. Nodes are not aware of the other nodes they are communicating with, or if anyone receives the messages sent. To receive data from another node, nodes subscribe to the relevant topic. Using the command `rostopic` in the terminal yields information about which topics exist, which nodes are subscribing and publishing to them, and what type of message types are used. Because topics are asynchronous, if synchronous messages are needed, it is often better to use services. In Python, a publisher and subscriber are created by writing:

```python
bool_pub = rospy.Publisher("topic_name", bool, queue_size =
    10)
rospy.Subscriber("topic_name", bool, callback_function)

bool_pub.publish(message)
```

The first line declares that the node is publishing messages of the type `bool` on the topic named *topic_name*. *Queue_size* is used to limit the number of messages in case a subscriber cannot receive them fast enough. The second line declares a subscriber to the same topic. Every time a message is received on the topic, the function `callback_function` is called and takes the received message as an argument. The last line is used to publish a new message to the topic.

Messages

Messages are naturally enough, messages, used to communicate between the nodes. They are sent over topics by a `publisher` in a node and received by a `subscriber`. A message contains simple data structures such as string, int, bool, float, and arrays of primitive types. They can also support arbitrarily nested structures and arrays. Structs in C are quite similar to messages. Messages can also be used to send fields of data and constants to help interpret the data field. Two or more nodes sending and receiving the same message must use a defined name of the message type. Otherwise, they will think they are using two different message types. This will also mean that one or both will not be able to publish or subscribe to a topic, as they are defined with a specific message type to be used. A typical message can look something like this:

```
Header header
int8 CHARGING_UNKNOWN = -1
int8 CHARGING_NOT = 0
int8 CHARGING_BULK = 1
int8 CHARGING_OVERCHARGE = 2
int8 CHARGING_FLOAT = 3
int8 CHARGING_BALANCE = 4

bool[] array_of_bools
int charging_state
float32 charging_percent
```

Initialising a message can be done by writing:

```
message = MessageType()
message.data = value
```

Where the first line initialises a message of the type *MessageType()* and the second line assigns a value to the argument *data*. In ROS there are many standard message types in the library `std_msgs`. When defining a message type, they are stored in `.msg` files in a subfolder called `msg/` in one of the packages using the message type. When doing this, the message files have to be added to the `CMakeLists.txt` file for that specific package. This so they can be translated to source code.

Services

Services are a method for synchronous communication, and are used when just sending a message and not receiving a response is not enough. They are especially useful in distributed systems. Service calls are blocking and therefore have to be short, like inquiring about a state or a value, or a quick calculation. Longer processes should not be done in a service call, and especially not processes that might be required to preempt. Services should neither depend on states or similar which can have unintended or unwanted side effects for other nodes. When a client calls upon a service, the service is executed and the response message sent back.

The request/reply is defined as a pair of messages in a `.srv` file where the request and response are separated by a line of "- - -". Srv-files are built using the same format as `.msg` files. An example of a service message can be something like this:

```
string placement
bool move
---
string result
string state
int8 speed
```

A providing node can offer a service by a string name, and a client node calls the service by sending the request message. Services also include the command-line tool *rossrv* which can be used to display information about the services and can be used the same way as *rosmsg*. The command-line tool *rosservice* is used for listing and acquiring information about services in a system, as well as dynamically invoking them.

To declare a service in a node the code in line 2 is used, like below. The callback function is called when the service is requested. The fifth line is a method for blocking while the service is unavailable, the sixth is the service handle and the seventh is the actual service call. The last line returns the response value.

```
1 # Service Node
2 srv = rospy.Service(service_name, ServiceType,
      service_callback_function)
3
4 # Client Node
5 rospy.wait_for_service('service_name')
6 service = rospy.ServiceProxy('service_name', ServiceType)
7 result = service(request)
8 returm result.response
```

Parameter Server

ROS parameters are used to create global settings and values. The parameter server is created and run automatically inside the ROS master. It is essentially a dictionary containing all global variables, which are accessible for all nodes. The parameters allow for testing or running a node without having to change variables all over the code, and for nodes written in C++ without having to recompile. All nodes can, at any time during runtime create, read or modify a parameter. The parameter server is not designed for high-performance and is best used for static data. Parameters are named using the ROS naming convention with hierarchy and strings, similarly to directories in Linux. Below the parameter */camera/left* is a dictionary containing *name* and *exposure*. The same would go for */camera*, which would be a dictionary of dictionaries.

```
# example of parameter names
/camera/left/name: left
/camera/left/exposure: 2.4
/camera/right/name: right
/camera/right/exposure: 1.8
```

The command-line tool *rosparam* provides commands for setting and getting ROS parameters on the parameter server using YAML-encoded files. In addition there are built-in functions for getting, setting and checking for parameters, `rospy.get_param`, `rospy.set_param` and `rospy.has_param`, respectively. Parameters can also be set in launch files, as in this example from [37], section 7.4:

```
<param name="param_name" value="param_value" />

<group ns="duck_colors">
    <param name="huey" value="red" />
    <param name="dewey" value="blue" />
    <param name="louie" value="green" />
    <param name="webby" value="pink" />
</group>

<node .... >
    <param name="param_name" value="param_value" />
</node>
```

The first line assigns the value to the parameter, and the second part creates a dictionary with four variables. It is also possible to create private parameters for one node. To do that, the first line is added to a nodes part in the launch file like in the last section above.

Action Server

The action server is a way to implement a standardised interface for interfacing with preemptable tasks and is provided by the `actionlib` package. Their most important characteristic is that they are preemptable, which has to be implemented cleanly by the use of action servers. These tasks are longer running and often provide feedback while running, like object detection, navigation, and laser scans. Slow routines or calculations which take more than a few seconds to finish, non-blocking background processes, or lower level controls are also examples of tasks best implemented by actions. It is also possible to execute more than one action goal in parallel on the same server. This means that the actions can keep each of the states for the lifetime of a goal, and the goals have their unique id.

Actions use the same principle as services and topics, they communicate via ROS messages. The ROS Action Protocol is built on top of ROS messages, and the client and server provide an API to request or execute goals by using functions and callbacks.

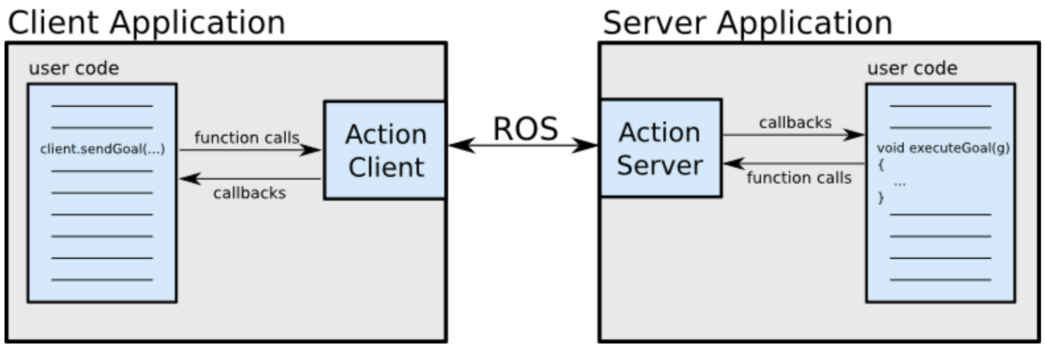


Figure 2.7: Illustration of how the client and server communicates in an action server. Image courtesy of [39].

Action messages are stored in *.action* files and are divided into three: *goal*, *feedback*, and *result*. The goal can be a position, the direction a camera is to point, or a state and is sent from the client to the server. The feedback is a way to tell the client about the progress of a goal. For navigation to a position, this feedback could be the current position. The result is sent when the goal is completed but is different from the feedback in that it is only sent once. In addition to these three topics, the protocol also sets up two others: *cancel* and *status*. These are predefined in the action protocol. *cancel* is used by clients to cancel goals, and *status* is used by the server to send updates about the goal to the client. A generic `.action` file might be something like this:

```
# Goal
goaltype goal
---
# Result
resulttype result
---
# Feedback
feedbacktype feedback
```

An ActionServer can be created like below, where examples from the Cyborg code are used. The last two lines set the calculated result in the result message, here a string, and then sets the result of the action and publishes it to the client.

```python
# include the result and the feedback messagetype
from cyborg_controller.msg import StateMachineAction,
    StateMachineResult
import actionlib

# create the resultmessage
result = StateMachineResult()

# create and start the server
server = actionlib.SimpleActionServer("server_name",
    StateMachineAction, execute, auto_start = False)
server.start()

def execute(self, goal):
    # action implementation

    # set result message and server state
    result.resultdata = "action succeeded"
    server.set_succeeded(result)
```

An ActionClient is created like this:

```python
# include the result and the feedback messagetype
from cyborg_controller.msg import StateMachineAction,
    StateMachineGoal
import actionlib
```

```

# create a client and connect to server
client = actionlib.SimpleActionClient("server_name",
    StateMachineAction)
client.wait_for_server()

# create and send goal with callback function to execute when
     completed
goal = StateMachineGoal()
client.send_goal(goal, completed_callback)
```

2.5 PAD Emotional State Model

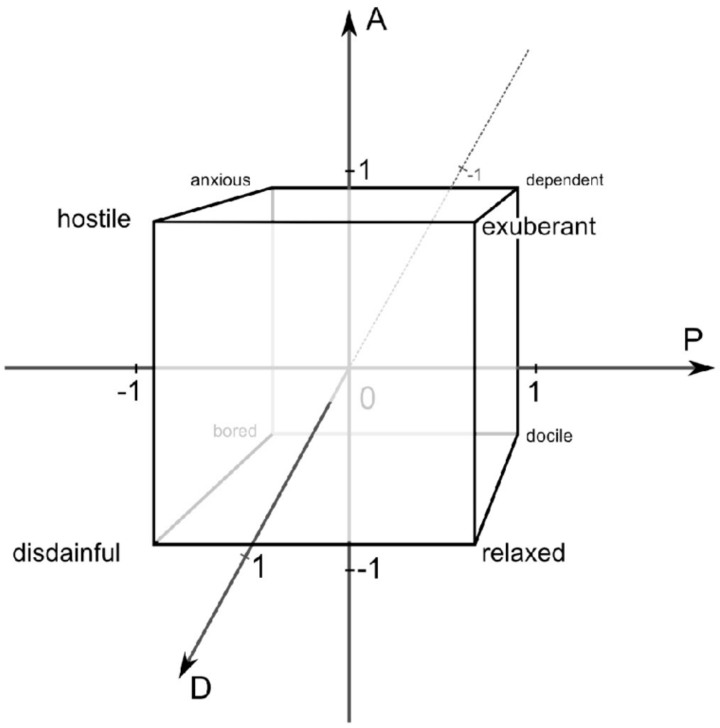


Figure 2.8: The PAD emotional model. Courtesy of [2].

Modelling human emotions can be tricky, and there exist many different methods for emotional classification. Emotional classification means a way to distinguish different emotions from each other. It is a widely discussed

Emotion	**Pleasure**	**Arousal**	**Dominance**
Angry	- 0.51	0.59	0.25
Bored	- 0.65	- 0.62	- 0.33
Curious	0.22	0.62	- 0.01
Dignified	0.55	0.22	0.61
Elated (Happy)	0.50	0.42	0.23
Hungry	- 0.44	0.14	- 0.21
Inhibited (Sadness)	- 0.54	- 0.04	- 0.41
Puzzled (Surprised)	- 0.41	0.48	- 0.33
Loved	0.87	0.54	- 0.18
Sleepy	0.20	- 0.70	- 0.44
Unconcerned	- 0.13	- 0.41	0.08
Violent	- 0.50	0.62	0.38
Neutral	0	0	0

Table 2.1: Some common emotions represented by the PAD emotion model's values from [29]

topic between researchers in affective science and emotional research, but there are two main schools of thought. The first that emotions are discrete, meaning every person has a set of basic emotions that are recognisable no matter their background or culture. Think of Mr. Bean, which is recognised all over the world despite the lack of conversation, but still fully comprehensible. The other school of thought is that emotions are characterised by different dimensions. This was coined by the father of modern psychology, Wilhelm Max Wundt. In 1897 he proposed the three dimensions pleasurable — unpleasant, arousing — subduing, and strain — relaxation to describe emotions.

The PAD model was developed by James A. Russel and Albert Mehrabian in 1974. It belongs in the dimensional models of emotion, and thus uses three numerical dimensions to represent emotions; pleasure, arousal and dominance. The pleasure scale signifies how pleasant an emotion is. Anger would score low on this scale, while happiness would score high. The arousal scale corresponds to the intensity of an emotion, and refers to a combination of mental alertness and physical activity. Here anger would score high, while rage would score even higher and happiness lower. Ecstasy is an example of a positive feeling that would score high on the arousal scale. The last scale, dominance, represents how dominant and controlling an emotion is. It can also be thought of as representing the 'fight or flight' response. For this scale, anger would be on the dominant side of the scale, while fear would lie on the submissive side. This domain also gives the possibility to differentiate between fear and anger, which, When compared to the Circumplex model with just two dimensions arousal and valence, look the same as they score high on both scales.

As seen in fig. 2.8, the PAD space can be divided into eight sub-spaces: hostile, exuberant, relaxed, and disdainful, and their counterparts docile, bored, anxious and dependent, respectively. There exist different names for each of these combinations of P, A and D, but they are all more or less synonymous. The axes remain the same either way, ranging from -1 to 1.

2.6 Finite State Machines

Finite state machines are methods of control and decision making based on states. It is in only one state of any number of states at a time, and transition from state to state depending on inputs. The defining trait of finite state machines are that the building blocks are states, all defined before execution. There are many ways to represent a FSM, the most common being UML state machines or event tables. There are two main types of state machines: non-deterministic and deterministic. Where each transition is uniquely identifiable by the input and source state and an input is required for a transition in deterministic state machines, non-deterministic state machines may have a choice of resulting states from a specific input and source state. The non-deterministic property comes from the fact that it is not possible to accurately predict which state the execution will finish just based on input and state.

2.6.1 Concepts

`States` are the descriptions of the status of a system. States can contain code to be run to perform some task or they can be waiting states waiting to execute a transition. If a condition is satisfied or an event is encountered

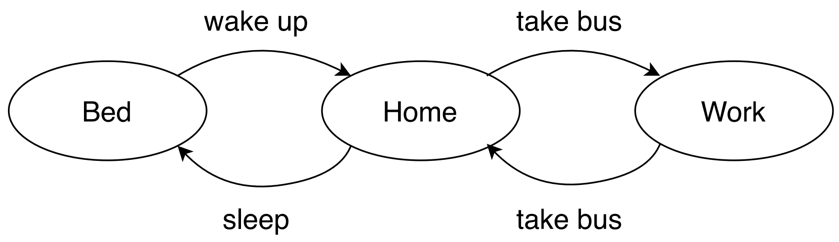


Figure 2.9: An example of a very simple state machine.

a `transition` is executed. Sets of `actions` make a transition, and can be performed when entering and/or exiting a state. `Events` are the driving force in most state machines, and tell the system if something has happened and it needs to change state. If implemented correctly they behave similarly to interrupts instead of blocking the system by polling. An example of an event can be that the system arrives at a location and should then perform some task. Here the event would be the arrival at the location.

2.6.2 Trade offs

Finite state machines act as a good firewall in that they protect from reaching unknown states, as all states and transitions are preset. They are relatively easy to understand as long as the system is not too big and complex, and they should be used with care in those instances. They are, however, a good way to break down behaviour, and clarify how to get to and from a state.

FSMs are not particularly compatible with scalability. If something is changed or a state is added, the logic has to be re-defined. As an FSM control system grows, the number of transitions, states and logic can quickly become unmanageable. Usually, an FSM uses a switch-case or if-else statements to implement the states. In an industrial setting where a system can have hundreds of states. This will result in a very high number of value checks and inefficient resource use. They are neither particularly compatible with concurrency. Running two or more state machines in parallel quickly leads to deadlocks or a need to edit the system. FSMs are also unsuited for unstructured behaviour like the Cyborg is expected to do. The most prominent behaviour being roaming around Glassgården. If the behaviour cannot be broken down into states or the number of states is not finite, then state machines are difficult to use.

2.6.3 In ROS

In ROS and the software on the Cyborg a task-level state machine architecture called SMACH is used to implement state machines. SMACH is a Python library installed in ROS to rapidly create complex hierarchical state machines. A drawback with SMACH is that it is not ideal for handling unstructured tasks. A further explanation of SMACH can be found in section 2.8.

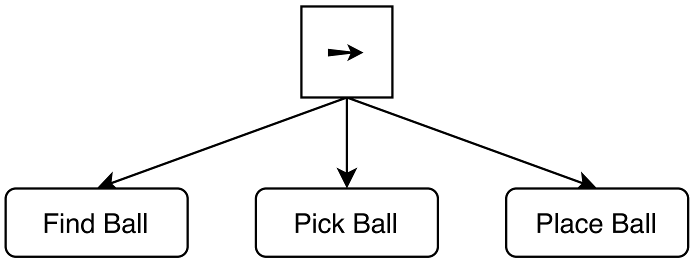
2.7 Behaviour Trees

Behaviour trees are mathematical models used for controlling behaviour and are often used in artificial intelligence and video games. They built in a modular fashion which allows them to describe the changes between a finite set of tasks. Very complex tasks can be created using simple tasks, without the need to know how each simple task is implemented. They are fairly simple to understand for humans which make them less prone to developer errors. Unfortunately, there does not seem to be a naming standard for the nodes in a behaviour tree, and depending on which implementation you are using the same type of node might be called different things. In contrast to hierarchical state machines, the building blocks are tasks and not states. This in turn creates other difficulties mentioned in more detail in section 2.7.2. Behaviour trees differ from Decision trees in that they are made for controlling behaviour where the latter is just for making decisions.

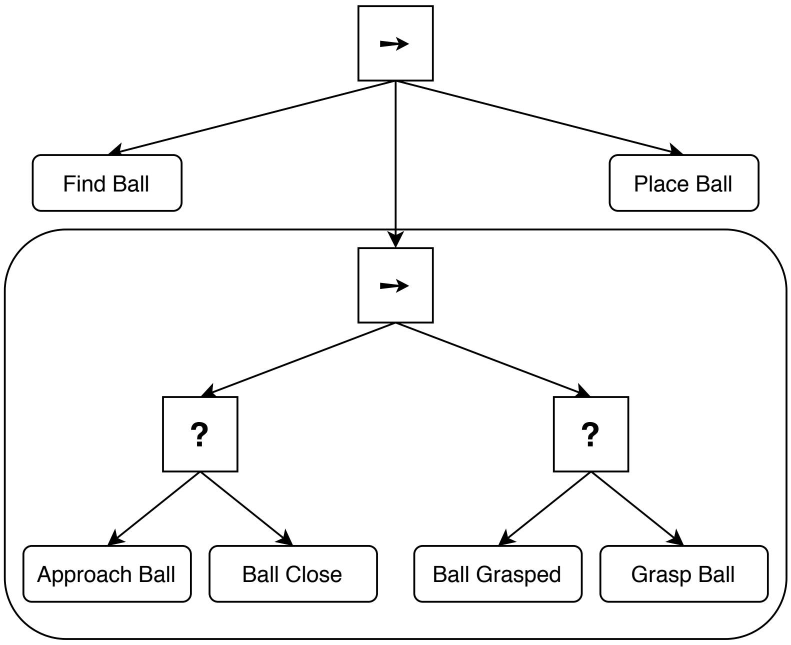
Decision trees must be implemented such that the children of a parent represent all possible outcomes for that node. If a node only can be answered with *yes* or *no*, then it must have two children. Decision trees can be part of a behaviour tree, becoming a sub-tree. Behaviour trees are evaluated from the root after they are reset, and each child is evaluated from left to right. The child node to the left has the highest priority, and its conditions are met it is executed. Because this child node is now set to 'running' the next time the three is evaluated it knows where to continue from. If any condition fails, it returns to the parent and then the next child.

2.7.1 Key Concepts

The behaviour trees are directed trees with nodes classified as `root`, `control flow` or `execution` nodes. The first have a single child, the second has one parent and at least one child, and the last has just a single parent. Children return either *running*, *success* or *failure* depending on if a goal has been achieved. The trees are *ticked* meaning at each tick is progressed down the



(a) A higher lever tree performing three tasks: finding a ball, picking it up, and placing it.



(b) In the sub-tree 'Pick Ball' the task 'Approach Ball' is run until the ball is appropriately close, then the task 'Grasp Ball' is run until the ball is grasped.

Figure 2.10: Two illustrations of different detail levels in a behaviour tree. Courtesy of [12] fig.1.1

tree depending on the node types.

Control Flow

Control nodes consist of two types `fallback` and `sequence` nodes. The fallback nodes, also called selector or priority nodes, are a composite type node and find and execute the first child that does not fail, and returns the status of its child. Fallback nodes are signified by the ? symbol. Sequence nodes will execute its children until one fails. They use the symbol $\rightarrow$. It will return a status code of running or failure depending on the return of the child.

There also exists a third type, `parallel` nodes, which use the $\rightleftharpoons$ symbol. They tick the children in parallel and return success if M $\leq$ N, and return failure if N - M + 1 children return failure. Otherwise it returns running.

Actions

Actions nodes are the nodes that execute some task and are often called leaf nodes. They change the state of the system and return running, success or failure depending on the execution. In fig. 2.10 the tasks 'Approach Ball' and 'Grasp Ball' are examples of actions. They can be longer processes and can be preempted.

Conditions

Conditions are checks to determine if a condition has been met. They are technically actions, but do not return running or change any internal states or variables of the tree. Another name often used are decorators. Examples of this in fig. 2.10 are 'Ball Close' and 'Ball Grasped'.

2.7.2 Trade offs

All tasks are broken down into their simple behavioural components, giving behaviour trees a new perspective on programming behaviour. This method is often less challenging to understand, regardless of background. A graphical interface for visualising and composing a tree allows for easier understanding and implementation for people unfamiliar with the inner workings of the system or non-programmers.

One limitation of behaviour trees is that they make it difficult to implement state-based behaviour and different operational modes where they do almost the same. This would require the implementation of several very similar trees or that each task must check which mode to run, meaning a state machine within each task. Behaviour trees do not replace the functionality of finite state machines, and therefore combining FSMs and behaviour trees can be a good idea. Though, it does increase the complexity of the system design.

The Blackboard

To share data between tasks, each behaviour tree has a data structure called a *blackboard*. It holds variables and values relevant to that tree. An example on the Cyborg could be a task fetches the current position which is needed for a calculation in another or a condition check. Depending on the implementation only some nodes may have access to the data.

2.8 SMACH - State MACHine

SMACH is a task-level architecture for creating complex robot behaviour. It is an open-source Python library and can be used to construct concurrent and hierarchical state machines. SMACH is useful when you want a robot to execute a complex plan of which all states and transitions can be explicitly described [44]. It allows for fast prototyping, complex state machines and introspection. If your system has unstructured tasks, is low-level or, as they state on their documentation page, when you want to smash something, then SMACH is not the best option. In and of itself, SMACH is independent of ROS, but the `executive_smach` stack allows for integration with ROS. It also includes integration of the `actionlib` package, which is a part of ROS.

SMACH has two main interfaces, *Containers* and *States*. Containers are a collection of one or more states and are the implementation of execution policy. The containers have a flat database to pass data between and coordinate states. The states in a container are stored like a dictionary. The most important job of the container is to define the transitions between states and what to do if a state wants to preempt another. *States* can mean different things depending on the context. In SMACH, a state corresponds to a state of execution of some task and all potential outcomes of that execution. The states are different from traditional state machines in that they describe what the system is doing locally, not the configuration of transitions between states. A SMACH state machine can be nested, meaning that one state machine can be a state in itself.

2.8.1 Creating a SMACH State Machine

The following is an example of how to implement a SMACH state machine with two states:

```
1  sm = smach.StateMachine(outcome = ["outcome3", "outcome4"])
2  sm.userdata.sm_variable = 0
3
4  with sm:
5     smach.StateMachine.add("State1", State1Func(some_function,
       arg1),
6               transitions = { "outcome1":"State2",
7                               "outcome2":"outcome3"},
8               remapping   = { "state1input":"sm_variable",
9                               "state1output":"sm_variable"})
10
11    smach.StateMachine.add("State2", State2Func(),
12              transitions = { "outcome1":"State1"},
13              remapping   = { "state2input":"sm_variable",
14                              "state2output":"sm_variable"})
15
16 outcome = sm.execute()
```

The first line declares a state machine with the name *sm* and its outcomes *outcome3* and *outcome4*. The container is made on the fourth line, and adds the states *State1* and *State2* on lines 5 and 11, respectively. Transitions are naturally enough the transitions from one outcome to another outcome or state. Remapping is not necessary but a good practice. It makes it easier to follow the flow of the code and not confuse variables used in states and state machines. To implement *State1* from the previous example you inherit the *state.State* base class:

```
1 class State1Func(smach.State):
    def __init__(self, function_inputted, arg):
        #state initialization
        smach.State.__init__(self,
                outcomes = ["State2", "outcome3"],
                input_keys = ["state1input"],
                output_keys = ["state1output"] )
        self.var = arg
        self.func = function_inputted

    def execute(self, userdata):
        #state execution
        if userdata.input == 1:
            return "outcome1"
        else:
            return "outcome2"
```

The class is initialised in the first `__init__` function and the state is initialised in the second. There the outcomes, input keys and output keys are set. The initialisation functions must not block further execution. Any other variables the state uses are set after the second, but still in the first initialisation function. The `execute` function is where the behaviour of the state is implemented. `Execute` can block for as long as needed. When returning from this function the current state finishes executing.

Chapter 3

Reevaluation of the Cyborg

The main goal of this thesis was to facilitate improvements to the behaviour module of the Cyborg. Though there has been done work with the controller and behaviour of the Cyborg before, it is lacking in some areas. These modules have been implemented using FSMs with SMACH which, as mentioned in section 2.6.2, have several drawbacks. Mainly the emotion implementation only had a few modes and needed tuning. When running tests at the start of this work, the Cyborg would switch between different animations and visualisations very quickly, and mainly stay in the idle state without doing much. In the behaviour launch file, there were many behavioural presets but only a small number would run.

During the work in the spring of 2019 [5], the use of behaviour trees was evaluated after looking at the work done in 2018 [30], but it was decided against using it as they were deemed too complicated to familiarise oneself with. It is the author of this thesis' opinion that both solutions are complicated and needs time to get to know. As a new student starting their work on the Cyborg would need to familiarise themselves with the whole system anyway, learning about behaviour trees should not be much extra work. Both implementations have some form of visualisation tool which makes use easier. The behaviour tree implementation used in 2018 included an editor and a run-time monitor of the system state, while the SMACH implementation has a run-time monitor but no editor. That being said, the current state machine has not been abandoned, as any new implementation of behaviour tree most likely will need some time to mature.

In addition to work on the behaviour module, work on other modules was needed to bring the Cyborg closer to the short and long term goals. Previously there have been implementations of object detection and computer vision which used a Xbox Kinect sensor, but they were either removed or not integrated into the Cyborg. A stereo camera from Zed and an Nvidia Jetson TX1 were already bought but not integrated as the software was not there. A wish to integrate object detection properly to use in navigation and behaviour had been expressed and has therefore been worked on this semester. Work with object detection is mentioned in more detail in section 1.1.

A wish to remove the Mode Selector Box and move its functionality into the ROS system has also been expressed. There was also a need for a monitoring system for the Cyborg. Implementation of a website is also being worked on. It can be used as a way to control and monitor the Cyborg, show a video feed from the stereo camera, check variables such as the emotional and battery state, place a point in the map and have the Cyborg move there, and look at historical data. The website will contain the Mode Selector Box's functionality and therefore act as a replacement, integrating its functionality into ROS on a remote computer.

After MobileRobot's closing down, their proprietary software will not be updated and poses limitations on functionality and future development. Therefore the ARNL package is being replaced by open-source packages from ROS. With the new navigation stack being worked on several minor changes has to be done in the rest of the system as many parts rely on or subscribe to the `ros-arnl` node. This includes changes from changing topic names to replacing functionality provided by `ros-arnl`.

Removing the Mode Selector Box and moving its functionality to the new website became the main reason to reevaluate the structure of the ROS nodes on the Cyborg. The nodes that interact with the hardware were kept as they were because they mainly have one or two topics they subscribe to where they receive what to do. That way, the mode would be set by using the website and then only the nodes needed would run. This will hopefully make it a little easier and more intuitive to understand, in addition to that the Cyborg does not have to be turned on and off to change the mode. In fig. 3.1 a proposal of the new system structure is shown. The main changes will be in the behaviour module, which includes the controller node, the event scheduler node, the primary stated node and the behaviour node.

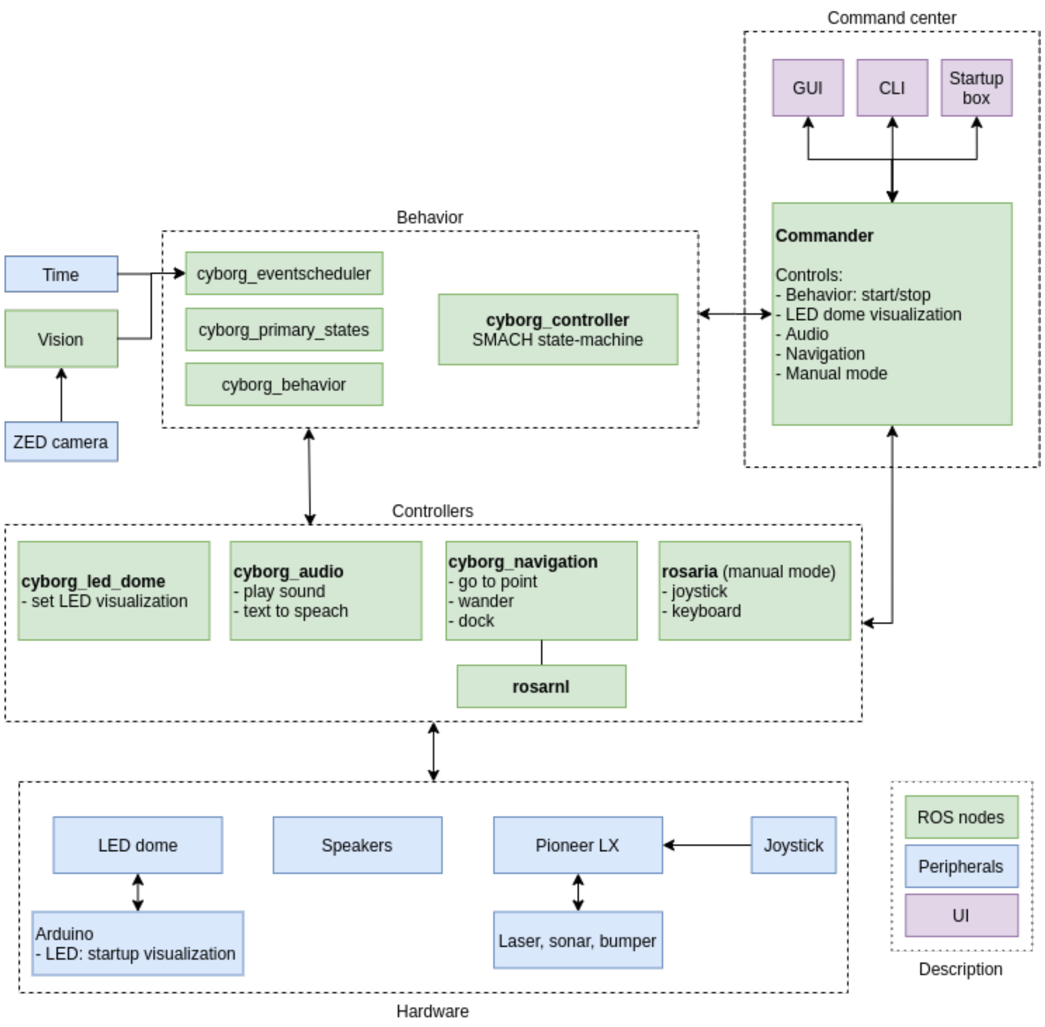


Figure 3.1: Proposed Cyborg software structure.

Finally, as the Cyborg uses software that is now several years old, and Python 2.7 is no longer supported many places since it reached its end of support date on January 1. 2020, it requires updating. The ROS and Ubuntu versions used on the Cyborg are also nearing their EOL dates, which provides a further incentive to upgrade the software. This has been investigated and is presented in section 7.1

Chapter 4

The Software Structure

As mentioned in chapter 3, a wish to replace the Mode Selector Box had previously been expressed. During the autumn of 2019, the Mode Selector Box' buttons and screen had stopped working, and because the cables in the box were kept in place by glue, it was decided to not try and revive it. The new website is the replacement for the Mode Selector Box and will be able to switch operation mode without needing to turn the Cyborg off and on again. This triggered a need to reorganise the structure of the Cyborg somewhat. The nodes that are communicating with hardware are kept as they are, while the nodes `event_scheduler`, `primary_states`, `controller`, and `behaviour` are organised into the module `behaviour` as shown in fig. 3.1.

4.1 The Cyborg's Current Structure

The main parts of the cyborg that are used in the behaviour module are presented below, with a description of the nodes. They are implemented as ROS nodes, and are using ROS' methods for communication.

The main issue with the controller and behaviour of the Cyborg is that it is hard to create unstructured behaviour. Especially the wandering behaviour where it is required to wander around Glassgården and executing various combinations of moving around, playing audio, and running various visualisations on the LED-dome. In addition to these things, it cannot appear to repeat a cycle of behaviours indefinitely and has to avoid obstacles such as humans, walls, stairs, bins and an assorted number of other things

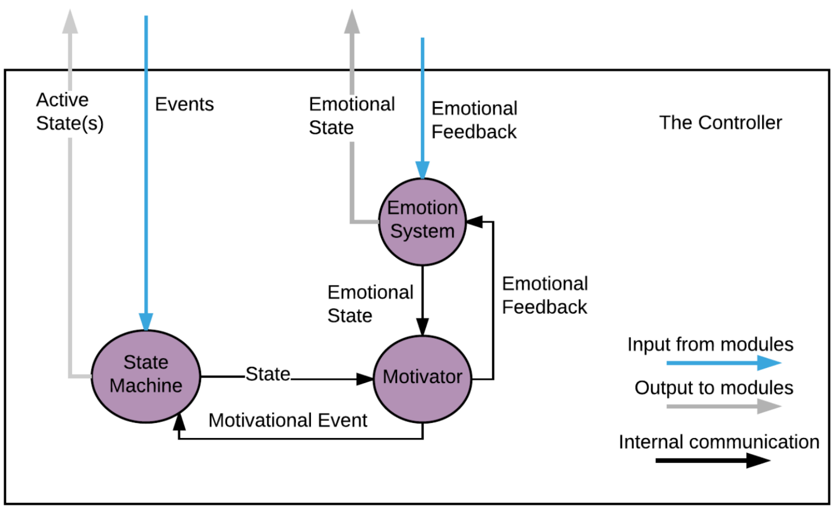


Figure 4.1: The original overview of the inner workings of the controller node [3]. This has been somewhat modified over the years, but the essence is the same.

that appear in Glassgården.

That is not to say that it is impossible. Because the behaviour is possible to break down into finite states, for now. But as mentioned, `wandering` presents the greatest challenge to this. But keeping the main goals and spirit of the Cyborg in mind, a way to implement the base of the Cyborg so that adding a feedback loop to the MEA data later becomes easier is favoured. This part of the Cyborg also suffers from the problem that it is big and complex, making it harder for new students working on it to add new implementations. The reason for this is ingrained in the concept of FSMs. Every time a state is changed or a new state is added, every excising state which interacts with the new has to be modified too. All this creates a need to rethink the way the Cyborg, at least, chooses what to do next. A deeper delve into the usage of behaviour trees can be found in chapter 5.

4.1.1 Controller

The Cyborg's state machine is implemented in the controller node, and it is responsible for coordinating all the other nodes. The three main parts of the controller are the motivator, the emotion system and the state machine.

The emotion system uses the PAD emotion state model to handle the emotional state of the Cyborg. The `motivator` is responsible for motivating

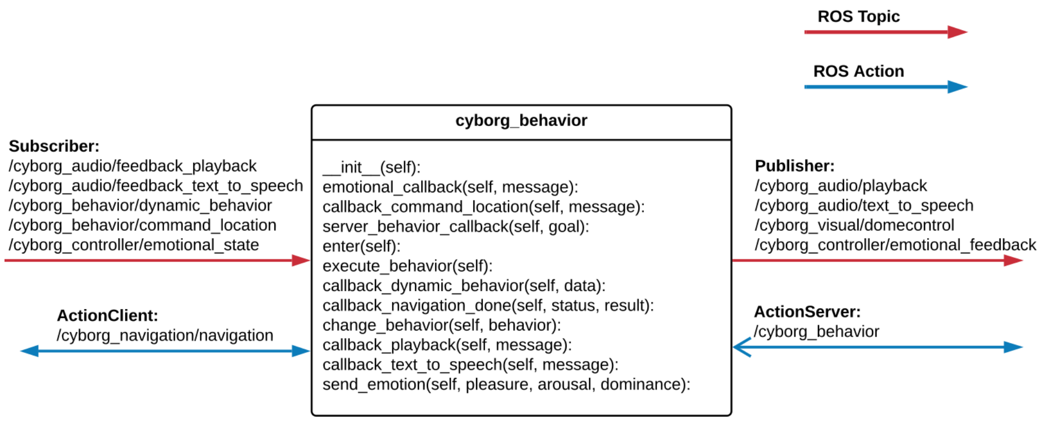

ROS Topic
ROS Action
cyborg_behavior
__init__(self):
emotional_callback(self, message):
callback_command_location(self, message):
server_behavior_callback(self, goal):
enter(self):
execute_behavior(self):
callback_dynamic_behavior(self, data):
callback_navigation_done(self, status, result):
change_behavior(self, behavior):
callback_playback(self, message):
callback_text_to_speech(self, message):
send_emotion(self, pleasure, arousal, dominance):
Subscriber:
/cyborg_audio/feedback_playback
/cyborg_audio/feedback_text_to_speech
/cyborg_behavior/dynamic_behavior
/cyborg_behavior/command_location
/cyborg_controller/emotional_state
ActionClient:
/cyborg_navigation/navigation
Publisher:
/cyborg_audio/playback
/cyborg_audio/text_to_speech
/cyborg_visual/domecontrol
/cyborg_controller/emotional_feedback
ActionServer:
/cyborg_behavior

Figure 4.2: Class diagram for the behaviour node class `BehaviorServer`. By [5].

the Cyborg to do activities when there are no external events. The goal of the motivator is to select actions that make the Cyborg happier. An in-depth explanation of the inner workings of the motivator and emotional system can be found in [3]. It explains the equations used to model the decay of the effect from the emotional feedback, the reward-based system, and its social cost of each action. The social cost keeps the Cyborg from repeating the same behaviour to accomplish the highest happiness value. Finally, the state machine is implemented using SMACH which organises all actions into states.

Figure 4.1 shows how the controller takes inputs in the form of events and emotional feedback from other nodes and returns the emotional state and system state. The action server in the controller is used by other nodes to publish events that can lead to a state change in the state machine.

4.1.2 Behaviour

The behaviour node serves as an interface for the output modules and a way to execute behavioural presets and simple states. It acts as an action server to be able to use the emotional state of the Cyborg to select which behaviour to execute and provide emotional feedback at the same time. The node includes methods for setting and changing audio and visual modes.

The node connects to the controller to provide emotional feedback and to get the emotional state of the Cyborg. The node's launch file contains

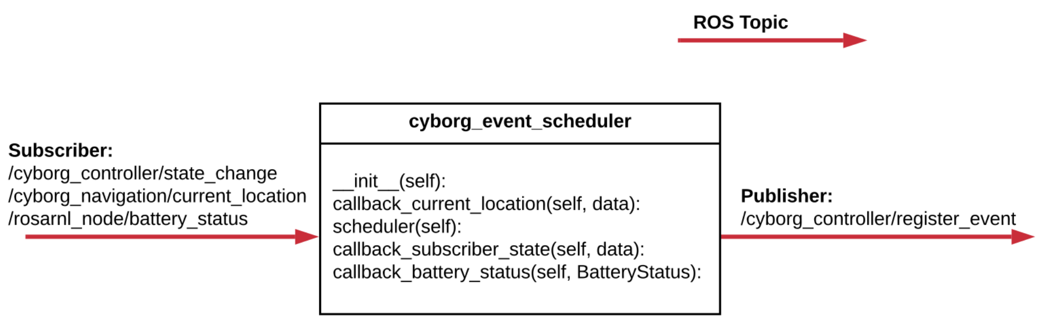

ROS Topic
cyborg_event_scheduler
Subscriber:
/cyborg_controller/state_change
/cyborg_navigation/current_location
/rosarnl_node/battery_status
__init__(self):
callback_current_location(self, data):
scheduler(self):
callback_subscriber_state(self, data):
callback_battery_status(self, BatteryStatus):
Publisher:
/cyborg_controller/register_event

Figure 4.3: Class diagram for the event scheduler node. By [5].

a database of behavioural presets. The presets can contain commands for visual, audio, navigation, and emotional feedback. They can define a completion trigger for the behaviour, and if the behaviour is dynamic. The trigger can either be the completion of audio, navigation, or the duration of execution, and callback functions are used to handle the feedback from the behaviours. Dynamic behaviours can change the behaviour by publishing a command to the node. If the Cyborg is moving from one place to another it may take some time, and during this, you may want to change the behaviour for it to avoid becoming tedious or repetitive.

4.1.3 Event Scheduler

The event scheduler node is responsible for functionality related to scheduled events for the Cyborg. It also detects and publishes system events like a low battery alert. This node can be used for alerting the Cyborg to any situations or conditions relevant to its behaviour, which are then communicated to other nodes using ROS protocols. The node subscribes to location data published by the navigation module, state data published by the state machine and battery status published by the `ros-arnl` node.

4.1.4 Primary States

The primary states node contains action servers that have more complex behaviour than those in the behaviour node. It also contains states like navigation planning, wandering emotional, idle, and states which do not use output modules. The node can execute state changes by publishing events

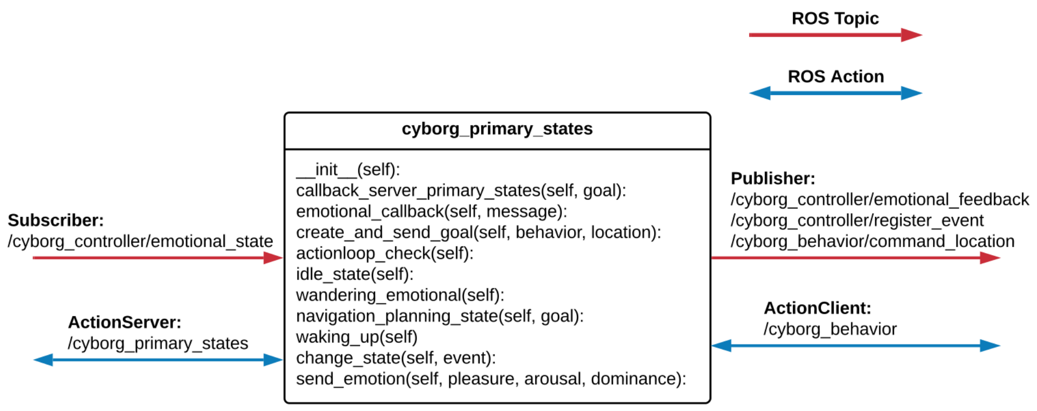

ROS Topic
ROS Action
cyborg_primary_states
__init__(self):
callback_server_primary_states(self, goal):
emotional_callback(self, message):
create_and_send_goal(self, behavior, location):
actionloop_check(self):
idle_state(self):
wandering_emotional(self):
navigation_planning_state(self, goal):
waking_up(self)
change_state(self, event):
send_emotion(self, pleasure, arousal, dominance):
Subscriber:
/cyborg_controller/emotional_state
ActionServer:
/cyborg_primary_states
Publisher:
/cyborg_controller/emotional_feedback
/cyborg_controller/register_event
/cyborg_behavior/command_location
ActionClient:
/cyborg_behavior

Figure 4.4: Class diagram for the primary states node. By [5].

and provides emotional feedback to the controller. All states provided by the primary states node uses the same action server and the callback function to execute the goal's corresponding function. For any states that use output modules, the behaviour node is also interfaced.

Wandering emotional activates a wandering behaviour. If the emotional state of the Cyborg changes and the new state is neither 'curious', 'unconcerned' or 'bored', then the state preempts itself. Navigation planning is executed when the Cyborg state machine receives a 'navigation_emotional' or 'navigation_scheduler' event. Based on the current emotional state it selects where and how the Cyborg should move, and then provides emotional feedback and initiates the state change.

4.1.5 Navigation

The navigation node provides navigational behaviour to the Cyborg. It can receive a position to move to and will then work out how to get there. The node has a common action server for all its states, and when an action goal is received it is parsed by the action server callback and the goal's corresponding function is executed.

The new navigation stack from this spring has replaced the `ros-arnl` node previously used with a variety of nodes from the ROS library. At the same time, the navigation node created for the Cyborg has been kept as unchanged as possible to keep from having to change other nodes due to the updates.

4.2 Previous Work with Behaviour Trees

In spring of 2018 work was done to explore the use of behaviour trees to implement a controller for the cyborg, among other work [30]. These are the parts of that work this thesis uses as a starting point. Firstly, different implementations and libraries for behaviour trees were evaluated. One of the main criteria was that they had to be written in C++ or Python as ROS only allows those two languages. Therefore the three final alternatives were *Pi Trees* [16], *ROS-Behavior-Tree* [11] and *behavior3py* [38]. Only the last was not created for use in ROS but was the only that included a graphical editor. When using Pi Trees or ROS-Behavior-Tree you had to implement the tree in the code, which makes debugging that much more complicated. The editor for Behavior3 creates a JSON file which is then exported for use with the code and ROS.

It was decided to use behavior3py, and a node called *cyborg_ bt* was made to use the library in ROS. In addition, a node for organising the custom decorators and actions was made, called *cyborg_ bt_ nodes*. When running the *cyborg_ bt* node imports the required decorators and actions into the cyborg control system and runs the tree created with *behavior3editor*. A behaviour tree node called `MoveTo` was created to enable sending navigation positions to the navigation module. Because the navigation stack has been updated and does not use `ros-arnl` anymore, this part of the behaviour tree implementation needs updating before it can be used. To monitor the system state during running a monitoring application called *rqt_ bt* was also created. It shows the structure of the tree and highlights the nodes that are being executed, and can halt and resume the execution of the tree. The application is implemented using *rqt*, which is a Qt-based framework for Graphical User Interface (GUI) development for ROS. It is often used to create and visualise graphs of the ROS nodes and topics like fig. 2.6 [50].

4.3 Behaviour Specifications

The cyborg has had many specifications, from more abstract like becoming a mascot for NTNU and the Department of Engineering Cybernetics to more specific like being able to navigate to a certain place in Glassgården. They give developers something specific to work towards and expose the requirements of the Cyborg. In the case of specifications for the behaviour module, they are limited only by our ideas and ability to implement them. Below the Cyborg's specifications for behaviour module are presented.

- The Cyborg should start the behaviour mode when receiving a start command from the website, otherwise it should stay in the suspension mode.
- If a system event is detected it should be processed and executed within a reasonable time.
- The Cyborg should have many different LED-dome visualisations, audio files, and movements which can be combined in different ways to create a library of behaviours.
- The Cyborg should be able to set a location to which it can navigate autonomously and it should be able to roam around Glassgården.
- If the battery percentage is below a set percentage, the Cyborg should abort whatever it is doing and return to its charging station.
- The Cyborg should alert the GUI to any changes in behaviour and continuously provide information about its location.
- It should be able to send messages to `domecontrol` and `audio` to change the visualisations and audio files that are played.
- The Cyborg should be able to either show off different visualisations on the LED-dome, play different audio files and move about without needing to change the state.
- The Cyborg should be able to execute a pre-programmed behavioural combination while being in one state, and change between these states after receiving commands from the website or internal events.
- The parts of the behavioural presets should start at the same time and run in parallel, not in sequence.
- The Cyborg should be able to use the MEA data to choose new behaviours and then execute them.

4.4 The New Structure

The updated structure of the Cyborg's software became virtually the same as in the proposed structure shown in fig. 3.1. There are currently three modes on the cyborg in addition to the startup called `behaviour`, `manual` and `demo`. The demo mode is intended for use at demonstrations where it cannot move and will then just play LED-dome animations and audio. Manual is intended to use the built-in possibilities to control the Cyborg using a joystick or the new website. The lower level nodes such as the LED-dome and audio nodes have been kept as they were, and the navigation has received an update. The audio and LED-dome node have different topics they receive their commands on and require nothing more. The navigation node is slightly more complicated but works in the same way.

To communicate with the new GUI website, a new node called `commander` has been made which works as a mediator between the GUI website and the rest of the Cyborg. The commander node must not be confused with the `command` node which is a command line interface for the Cyborg created in 2016. When the Cyborg is started and finished initialising it is in the suspension state until the commander receives a message to run one of the modes and it transmits an execute message to the nodes needed for that mode. If a message changing the mode of the Cyborg is sent from the website, the commander node transmits an abort message before running the new mode's respective nodes. Say the mode `manual` is chosen. Then `commander` will take commands receives from the website and transmit them to `rosaria` telling it to turn or move forwards or backwards.

Because of the new navigation stack, any usage of the node `ros-arnl` has been removed. Any messages published in topics subscribed to by other nodes and functionality that already existed in Aria has been replaced with those. Functionality not possible to replace using Aria still needs to be implemented again. This includes the built-in behaviour where the Cyborg would find its charger and set itself to charge if the battery was below a certain limit.

The main state machine and the nodes concerning it are collected in what is called the behaviour module. Which is what is run if the commander receives a command to start running the full Cyborg system. It must be noted that all ROS nodes are launched when the Cyborg is started, but those in the behaviour module remain in an idle state until they receive a run command from the commander. It is also this state machine which preferably would be, if not fully, then at least partially replaced by behaviour trees. Using both behaviour trees and state machines is especially useful if you want to represent alternate behaviour in different modes. For example, if the Cyborg was to behave differently in a low power mode than in the normal mode, having this combination would be helpful.

Chapter 5

The Behaviour Module

The behaviour module is the brain of the Cyborg and the module with the most development potential. The nodes it contains control the emotion system, handle external events, choose the next behaviour and combine them. Adding object detection to be Cyborg will open new possibilities for the behaviour module, the most important being able to detect people and execute different behaviours based on that. More active usage of the MEA data will also bring new possibilities to the Cyborg. It can be used to decide behaviours and as feedback to the emotional system. To be able to implement a feedback loop to the neural cells would be a big milestone for the project.

Currently, MEA data from file is only used to create the interpreters `moving_average` and `individual_moving_average`. That same file could be used to create emotional feedback by running it through various filters and not always reading it from the beginning. The file contains 10 seconds of data recording from the neural cells, equivalent to 100.001 lines of data, which should be enough to generate a large quantity of other data. Even though the contents of the file look mostly like noise. Additionally, there exists another file containing MEA data which is one second long.

During the work with this thesis, there were mentions of a desire to update or change the system the cells are grown in at St. Olav's by those responsible for that part. While this does not affect the work of students from the Department of Engineering Cybernetics directly, it would most likely mean that the part of the Cyborg's software which is responsible for communicating with the cells' server would have to be updated as well. This part of the software was developed by an EiT group in 2017.

5.1 Evaluation of Behaviour Tree Implementations

One question that emerged during the evaluation of and work with the behaviour module was if *behavior3py* was the best behaviour tree library to use. One advantage is that this is the library already used with the Cyborg, but it has not been used extensively yet. Behavior3py has not been updated since June 2015, and there are a number of other libraries that are both newer and regularly updated. It was originally written in JavaScript but was translated to Python. The issue with changing to another library is that all the nodes that use behavior3py have to the modified to function with the new library. Anyhow, this section will take a look at other behaviour tree libraries, their strength and weaknesses, and compare them to behavior3py.

Most behaviour tree libraries are created for designing NPC behaviour and have been used in games such as Halo, Bioshock and Spore. The question is whether or not they will be compatible with ROS, or how much work it will require to create a node for them to work with ROS. Though, the best option would most likely be to use a library already compatible with ROS. Other considerations to take into account are how convenient the libraries are to use. Do they have a graphic editor making it much easier to design a tree, and is it possible to visualise the tree during runtime making it possible to follow the flow of the tree.

In the paper *"A Survey of Behavior Trees in Robotics and AI"* [22] the authors collected some of the behaviour tree libraries available at the time (November 2019). There, Behavior3 is only mentioned with its JavaScript implementation. From this report, it is obvious that not many of them have a designated GUI editor or are created for communication with ROS. Even fewer can do both. The report also lists what language the libraries use and when their last commit was. All of them are open-source. The report affirms what research showed, that the most interesting libraries for the Cyborg were *behavior3py*, *BehaviorTree.CPP*, *ROS Behavior Tree* and *Py Trees*. Of those libraries, only the second and the last have recently been updated.

The most used behaviour tree library is *py_trees*, most likely because it is well maintained. It has been extended to *py_trees_ros* to include extensions for robotic implementations in ROS. Because it has been released for both

Name	**Language**	**GUI**	**ROS**	**Last Commit**
py_trees [45]	Python	-	√	14. May 2020
behavior3py [38]	Python	√	-	24. June 2015
BehaviorTree.CPP [6]	C++	√	√	11. June 2020
ROS-Behavior-Tree [10]	C++	√	√	22. October 2018

Table 5.1: The behaviour tree libraries most relevant for the Cyborg

ROS Kinetic and Melodic, it is relatively safe to assume they will have a release for ROS Noetic. There are also releases for ROS 2. The library has some restraints. It has no sharing of data or interaction between tree instances, trees cannot be run in parallel, and it can have only one behaviour executing or initialising at a time. Instead, information sharing between nodes is enabled by using a blackboard. The blackboard only works for one tree, meaning that if several NPCs were running they would not be able to share information. This should not be an issue for the Cyborg. Sequence nodes are only implemented with memory which removes one of the advantages of behaviour trees, namely the reactivity. Implementing a sequence node without memory should be fairly straightforward. The package *rqt_py_trees* provides a GUI plugin for visualising the trees. There is also a version for ROS2.

ROS-Behavior-Tree does not have a working implementation for the latest version of ROS, but is both compatible with ROS and has a GUI. It has a parallel control flow node which means it is possible to have parallel execution of leaf nodes. It features a helpful user manual. Because this library lacks support for the latest ROS versions this is not the library to choose if the decision to update the Cyborg to Ubuntu 20.04 and ROS Noetic, or their previous releases. *BehaviorTree.CPP* is another library written in C++. This is both well maintained and documented which makes using it very simple. It is also ROS compatible and features a GUI called *Groot*. The trees can be created at runtime using XML, and actions can be made asynchronous. Custom nodes can be linked statically or the can be converted into plugins which are loaded at runtime.

Behavior3py cannot run different action nodes in parallel, which will probably be needed on the Cyborg. One example where parallel execution is necessary is to be able to run the wandering behaviour and at the same time stop the current audio or interpreter, choose a new and execute that. The same way it is possible to implement a non-memory sequence node for *py_ trees* it is possible to implement a parallel composite node for *behavior3py*. To circumvent the missing parallel node, one possibility can be to create a sequence where the variables and states needed for a certain behaviour are chosen and stored using the blackboard. Then a last node in the sequence reads those from the blackboard and starts the correct ROS actions to execute the behaviour. It may require more work and be more cumbersome, than creating a parallel node. The emotion system runs in the background and to use the emotions to choose the next behaviour the blackboard has to have access to those variables, or the emotion system has to send them to the blackboard. It would be useful to have services attached to composite nodes and are run in the background as long as the composite node's branch is running. They can then be used to access the emotional system or retrieve status variables, which could then be written to the blackboard. This would simplify checking conditions and events in the behaviour tree nodes.

BehaviourTree.CPP or *py_ trees* seems to be the best options for the needs of the Cyborg. The only downsides being that one is written in C++ which is a compiled language unlike Python and the other lacking a GUI, respectively. Their main advantage is that they are well maintained and keep receiving updates. Having a way to visualise the tree, both while designing it and while its running is particularly helpful. Because behaviour trees become very intuitive and straightforward to read. Not having that possibility would be a shame. Then comes the question of is it worth changing to a new library. While there are not that many nodes or supporting code implemented for the Cyborg yet, it may be daunting to do this change for someone who is not familiar with the Cyborg or behaviour trees. And it will require a great deal of work. Because *behavior3py* could not initially communicate with ROS, a ROS node was created to retrieve and create the tree from its JSON file. If the decision to change library is taken, and this is to be done at the same time as upgrading the OS and ROS, it is important to remember that only the extensions to the libraries need to be created for the correct ROS version. The libraries are independent of ROS versions. Additionally, the behaviour tree nodes may not need to contain much ROS functionality. They might

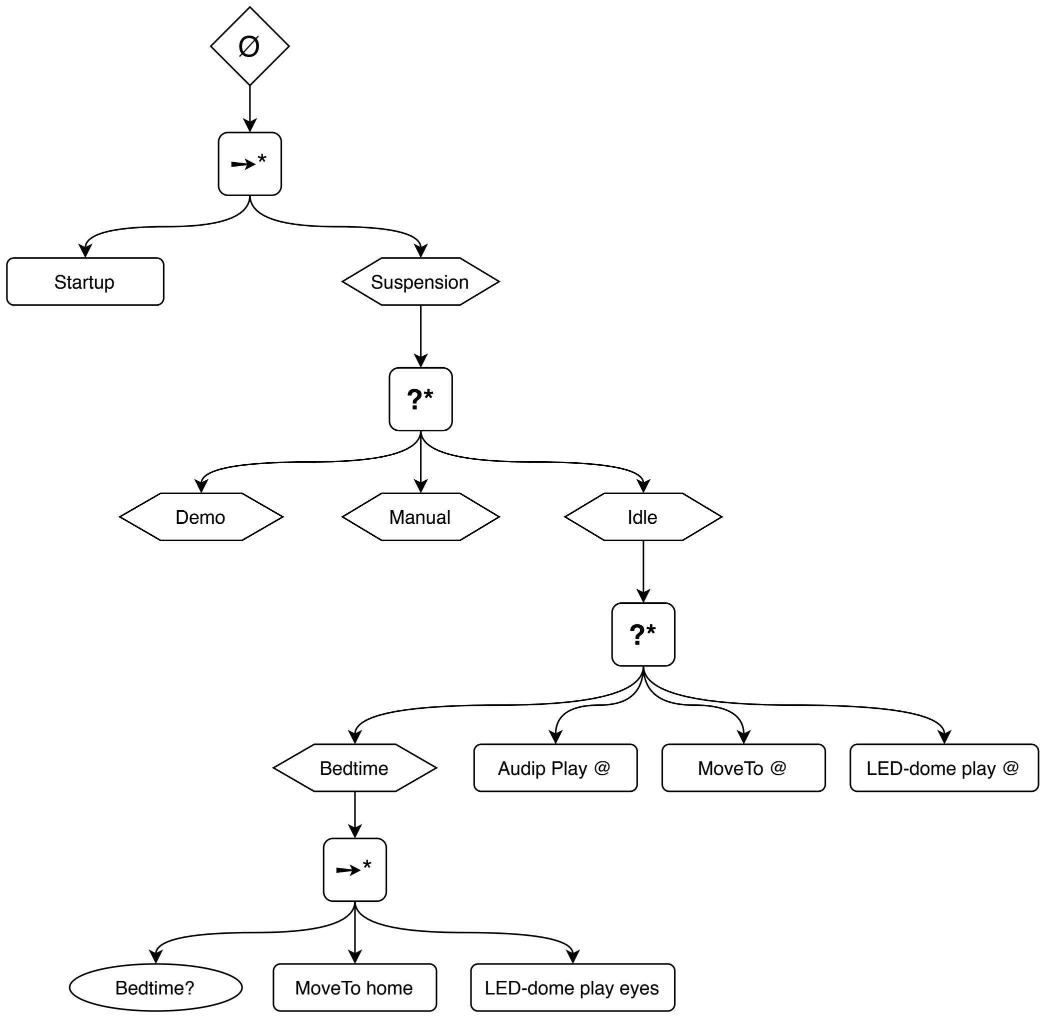


Figure 5.1: A proposed structure of the highest level of the cyborg behaviour tree, with some actions added to the `idle`-tree for completeness. The tree boxes are added for readability and clarity.

only need to send and receive messages, depending on how much functionality from the existing ROS nodes would be needed in the behaviour tree nodes. Ultimately, the decision has to be made by the future students working on the Cyborg. But if the decision to change is made, then *BehaviourTree.CPP* or *py_ trees* are the best libraries at this point.

5.1.1 Behaviour Trees and the Cyborg

Using *behavior3editor*, a high-level behaviour tree describing the Cyborg's behaviour at startup was created. While it requires more work for a complete implementation, it works as a starting point for future work. The behavioural presets of the Cyborg can be implemented as subtrees like *Bedtime* in fig. 5.1. A proposal is to create three lists, which contain LED-dome interpreters, audio files and locations, and the tree will choose one of each. This can be done using three selectors as children of a parallel node, but there are several possible implementations. They will then be executed in parallel just like any of the behavioural presets, either by using a parallel node or starting three ROS actions.

A shortcoming with both behaviour trees and finite state machines is the difficulty of representing different modes, like a normal power mode and a low power mode. The most obvious way to fix this is to use a combination of FSMs and behaviour trees, which would most likely be used regardless because of how the Cyborg is implemented. The question becomes how much of the implemented functionality should be moved to the behaviour tree nodes versus how much should be kept in the ROS nodes. The behaviour tree nodes not need to contain anything more functionality to send and receive messages to the ROS nodes to start actions or call a service. Additionally, keeping much of the functionality placed in ROS services and actions keep it as reusable as possible.

Figure 5.1 shows the implemented high-level tree of the Cyborg at startup. The hexagons represent the root of different trees and are created in their own file in the editor. This whole tree could be implemented as one tree without the subtrees, but dividing them into several smaller parts make it more convenient when you want to reuse parts. The * denotes that those are memory composite nodes, meaning they can access the blackboard. If there is no need for accessing the blackboard the standard composited are used.

5.2 Psychology of the Cyborg

The Cyborg is a fascinating study in, not only cybernetics but other fields as well. For example how different behaviours or things the Cyborg can do affects people and what it can do to grab people's attention. Humans are relatively unpredictable, and for the Cyborg to become more lifelike it has to become more unpredictable too. But to understand how to do that, it helps to study the psychology behind it.

The Cyborg is limited in its abilities to express itself using physical movement. Imagine examples like BB-8 and R2D2 from Star Wars and Pixar's Luxo Jr. lamp featured in their short Luxo Jr. and appears in their production logo before every feature film. Their movements and actions express intentions and emotions without language, except for sound effects. Pixar is exceptionally good at this. While R2D2 does not have the same freedom of movement as the two others it is still very much comprehensible. All these, and so many other similar characters from movies and television, provide excellent inspiration. Giving the Cyborg audio, in the form of sound effects will also aid in giving it life and emotion without language. There are two different options concerning sound. The first is to create new sound effects. The second is to use existing from examples like those previously mentioned. Using existing audio and sound effects may make observers think the Cyborg is pretending to be the original, but it requires much less work than creating one's own. Making it look like the Cyborg is pretending to be another robot or character from a movie might be perceived as lame, but most likely people will be positive.

5.2.1 Colours and Psychology

To help the Cyborg express emotions it would help to show facial expressions on the LED-dome, and while this might be difficult, it might help to use different colours. Either the facial expression is shown in a specific colour, that the rest of the dome is in that specific colour, or that the edge of the dome has that colour. Though, having the whole LED-dome in one background colour while the face is another colour can come with some complications. For example, for the face to be properly visible and distinguished from the background colour it will have to be a contrasting colour. LEDs lack the ability to create the colour black. Black comes from a lack of light. That is how OLED screens work. The contrasting colours may also have the effect that the face's colour is perceived as the colour corresponding to the emotion instead of the background.

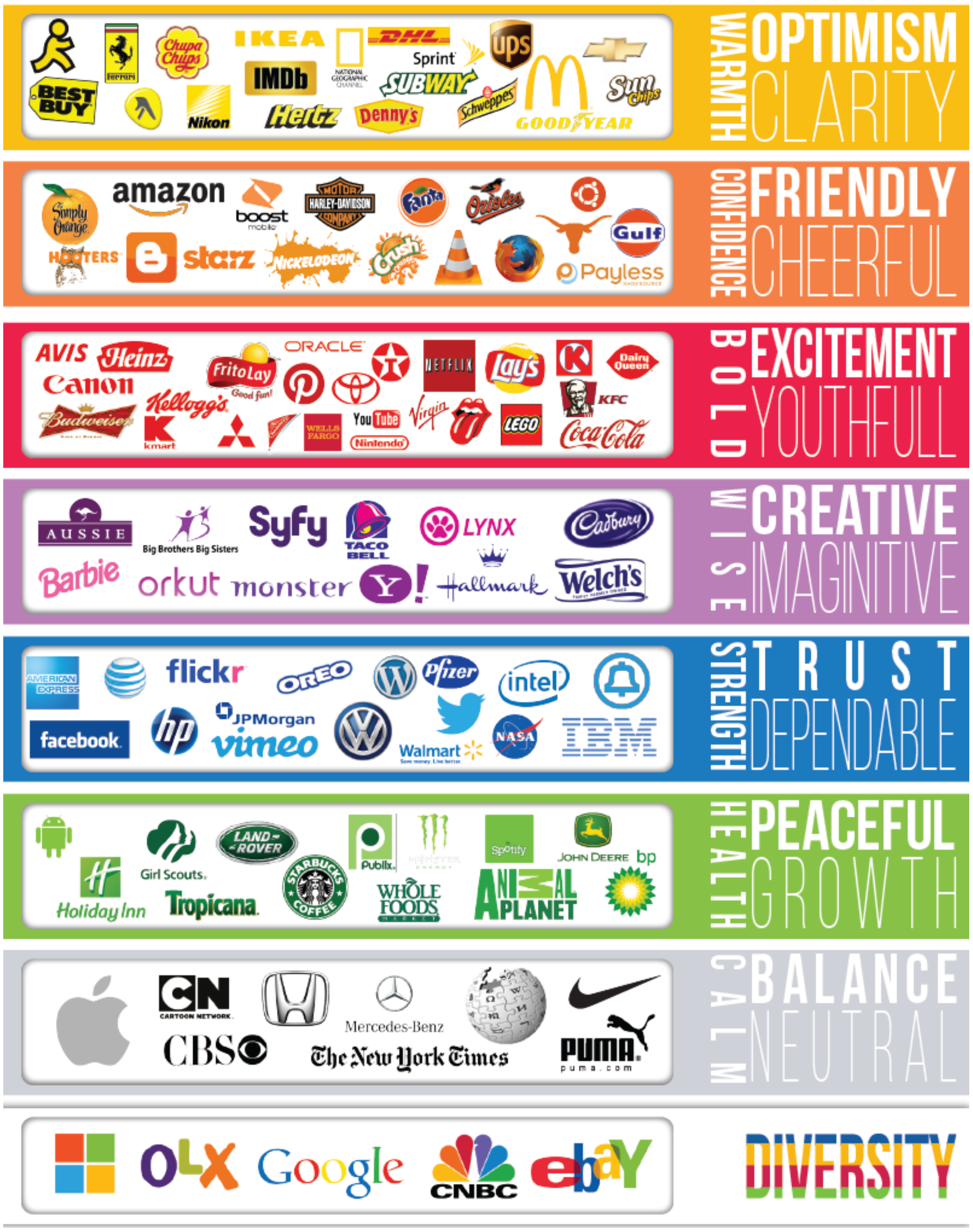


Figure 5.2: Colours and most used significations, and some of the brands that use those colours in their logo. Image courtesy of [28]

Colour theory is the visual effects of a specific colour combination and a body of practical guidance to colour mixing. Few things are more important, or more subjective, in design than the use of colour. Colours can bring different emotions. Most often they do this because of the use of colour in pop culture like cartoons, advertisements or brand's use of colour in their logos. These associations can vary depending on the culture people grew up in, evolution and personal experience. The villain almost always has a green colour scheme in Disney animated movies and character become red in the face when they are angry or green when they are sick. In nature poisonous animals usually have bright colours to warn and deter predators. Some animals take advantage of this evolutionary warning and adopt bright colours even though they are not poisonous.

In their paper *From Color to Emotion* [19] Donald Hoffman and Shannon Cuykendall write "We propose, then, that the connections between colors and emotions are few and weak, but that the connections between chromatures and emotions are many and strong." They introduced the concept *chromature*, which comes from combining *chroma* and *texture*. A chromature is a small image patch in a specific colour, typically not showing the entire object and typically not homogeneous in hue, saturation or brightness. They found that chromatures could evoke strong emotions even though the object was not recognised by the test subject. Which coincides with what many brand logos try to do. For simplicity's sake, colour has been used here instead of adopting the concept chomature.

Colours have been used to represent emotions in pop culture countless times, both for positive effect and negative effect. Think for example the colours of the houses in Harry Potter. Gryffindor is known for courage, daring, chivalry and a strong moral compass, and their colour is red. Hufflepuff is known for loyalty, dedication, honesty and humbleness. Their colour is yellow. Ravenclaw's strengths are intelligence, curiosity, creativity and individuality. Their colour is blue. Lastly, Slytherin is known for ambition, resourcefulness, determination and they are cunning, and their colour is green. The colours of the feelings in Pixar's Inside Out are the stereotype for what colours represent. The character Joy is yellow, Sadness is blue, Fear is purple, Disgust is green and Anger is, naturally, red.

Big brands are masters of using colour and shapes to their advantage. They use colours to bring out and play on certain feelings and emotions to get people to buy their products. Most people are unaware of the power colours have on consumers, and that is of course when they have the biggest impact. Spotify changed the colour of their logo from what they called "broccoli green" to a more "pop green". The new colour is brighter and cleaner, keeping the same tone instead of fading to darker or lighter. Instagram changed its logo from a more realistic looking brown Polaroid camera to a more abstract looking camera with several colours fading into another. When looking at fig. 5.2, Instagram could fall into the purple category. And, after all, there is a reason NTNU's logo is blue.

All these meanings and emotions colours can bring with them are a great way to add more life and impact to the Cyborg, and are a great starting point when choosing the colours of different interpreters. While it is not the logo of a brand, the same techniques can be utilised. Below are some examples of colours and their related emotions and what they can signify.

- **Red** - passion, strength, power, urgency, attention, alarm
- **Blue** - comforting, confident, professional, trustworthy, responsible
- **Yellow** - intellect, happiness, fresh, cheerful, positive, excitement
- **Green** - safety, wealthy, healing, peaceful, progressive, growth
- **Purple** - luxury, sophistication, mystery, ambition, spiritual, royalty, creativity
- **Orange** - enthusiastic, happy, creativity, determined, optimistic, excitement
- **Brown** - reliable, confidence, casual, natural, warmth
- **Pink** - romantic, feminine, fresh, fun, delicate, compassion
- **White** - simplicity, purity, safety, cleanliness, softness, innocense
- **Black** - bold, power, luxury, sleek, mystery, elegance, authority
- **Gray** - cool, neutral, authority

5.3 Adding A Behaviour to Behaviour Tree

Adding new behaviours to an existing behaviour tree is relatively easy. The trees are created to be modular like ROS nodes, meaning nodes can be added or removed without interfering much with the rest of the tree. To run the behavior3editor using the terminal, change into its directory and run `gulp serve`. The script *start_ behaviour3editor* can also be used. It was created to ease running the editor from a launch file.

To add a module to the behaviour tree using behavior3editor, you first have to import the existing tree into the editor. Trees and nodes can also be imported by themselves. Using the behaviour of how to choose what to eat when hungry, we will implement that using the behavior3editor. Depending on how high-level or low-level the behaviour is, the tree will be more or less complex.

Let us call it the hungry tree, shown in fig. 5.3, and we start by checking if we are hungry and then if we have any food. This would mean adding a *sequence* node with three children, where the two first are actions and the last a composite. Depending on if you need to access the blackboard or not the sequence node can either be a *sequence* node or a *memsequence* node. Say for simplicity's sake that we do not need variables from the blackboard. The last composite could be a sequence or a selector depending on what the behaviour is supposed to do. If it was to just cook some food and that cooking was divided into sub-actions, then it could be a sequence. We will make it a selector where it can select between three subtrees, *pizza*, *burger* and *pasta*. Those subtrees can be very similar except their nodes take an argument which is the food to eat.

Then we add a sequence node as the start of the pizza tree with four children: *eat pizza*, *cook pizza*, *make pizza*, and the subtree *gather ingredients*. If any of the actions fail, the sequence will fail. Nodes to try again or repeat for a certain number of times or until it completes exist to not make the tree just abort the whole behaviour. The subtree *gather ingredients* exist of a *open fridge*, a *find ingredients* and a *close fridge* action node. These actions can be broken down into even smaller parts depending on what is needed. Each action leaf node must have a method that can be executed. The method has to report whether it has failed or succeeded, and it should return running during execution. This *hungry* tree could be created as one

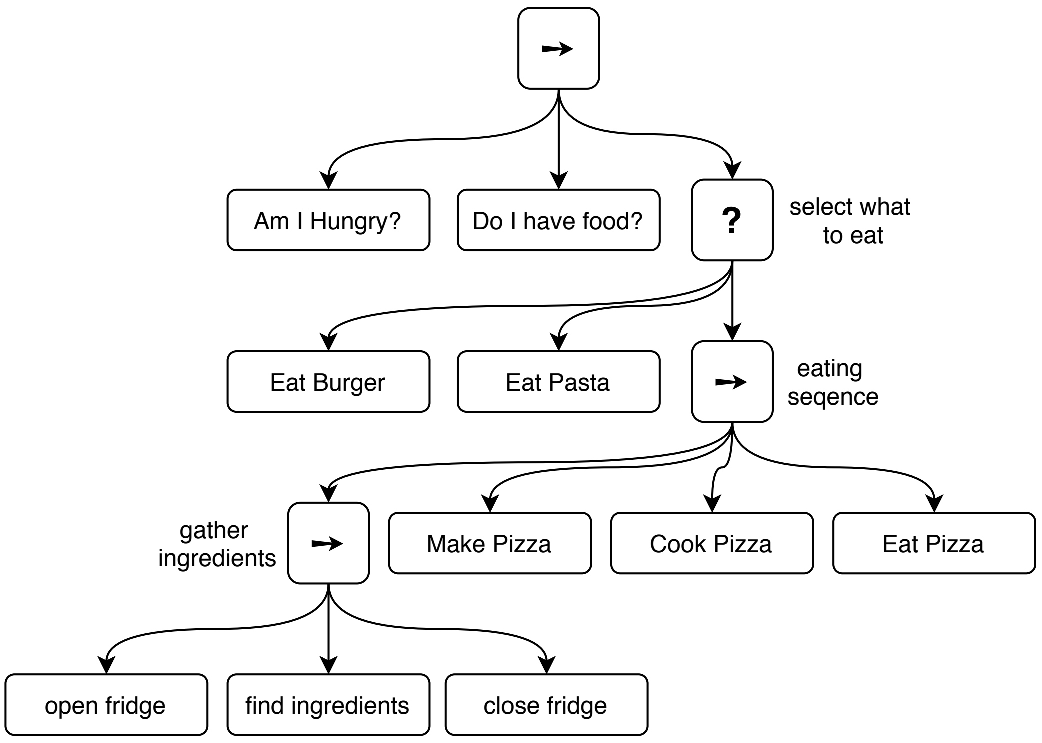


Figure 5.3: The behaviour tree *hungry* showing the flow of choosing what to eat.

tree, or be built up of several subtrees, which can be very simple like *gather ingredients* or more complex. To use several smaller trees in behavior3editor each tree has to be created on their own and then drag-and-dropped into its parent tree. It may be easier to keep track of the whole tree if the subtrees do not become too small.

5.4 Proposed New Behaviours

During work on the Cyborg, there have been many suggestions for what it could do to expand its behaviours. Some are silly, some are very complex and others funny, but they are a great source of inspiration. Many would fit perfectly for when the Cyborg encounters a person which it could detect using object detection, others can be played when it is wandering around, and some ideas came from working on collecting audio files. Using quotes from movies and television gives a fun twist to everything the Cyborg says and does, and we avoid recording someone. Which would mean that, in time, there the

Cyborg would have several different voices, or that everything would have to be re-recorded every year. You also evoke the same effect which BB-8, R2D2 or Bumblebee from Star Wars and Transformers also create. The proposals are presented below.

- When leaving someone or moving to a new location it would play "get to the choppa". A quote by Arnold Schwarzenegger from the movie Predator. This audio file has been added.
- While not entirely sure what to use the quote "say hello to my little friend" from Scarface for, there was a consensus that this would have been a fun addition. But the audio file was not added.
- When someone rounds a corner and the Cyborg is there and detects them, it could play some form of "boo!". This could give unfortunate results, but the Cyborg is neither fast, loud nor intimidating which is why it was deemed safe and kept as a proposition.
- When wandering the Cyborg could function as a speaker playing from a radio station. The radio station could be changed using the website.
- The behaviour or the Cyborg's mood could be controlled or influenced by MEA data or the current weather.
- Create race stripes on the LED-dome and drive around Glassgården playing sounds from race cars. This behaviour has been added to the behavioural presets. It could also contain a countdown before starting to 'race' around.
- When the Cyborg's battery reaches below a certain level it would change behaviour slightly to appear to be more tired, and when it decides to go to the charging station it could tell people it is going to sleep.
- While charging the Cyborg could either run the interpreter `charge` or `suspension`. The first was implemented last autumn and the latter this semester. The state it would be in while charging would not be *suspension* even though the interpreter is called that.

- The Cyborg could tell someone they meet the weather for tomorrow.
- In addition to running the interpreter *startup* and playing an audio file when turning on the Cyborg, it could also turn 360° to show that everything is initialised. It should not, however, do this while at the charging station.
- The Cyborg should be more active around 'akademisk kvarter' because at that time there are more people in Glassgården as they have breaks from lectures.
- During advent the Cyborg could play Christmas music, and play the quote "Merry Christmas, ya filthy animal" from Home Alone 2: Lost in New York. Though it is actually a quote from a movie created for and shown in Home Alone 2 called Angels with Filthier Souls.
- If someone cuts in front of the Cyborg while it is driving around, it could play "I'm walkin' here" from Midnight Cowboy
- To grab the attention of the public the Cyborg could play "Come quietly, or there will be trouble" from Robocop, or similar quotes.
- Create other behaviours using the newly added audio files listed in chapter B.

5.5 Finding PAD Values to Represent Emotional Feedback

To utilise the emotion system used on the Cyborg properly the behavioural presets need to have an emotional feedback to return to the motivator when done executing. The emotional feedback is what motivates the Cyborg to choose and execute the next behaviour, and it makes sense that most behavioural presets give this feedback so the Cyborg does not end up doing the same things or not doing anything at all. Previously only a handful had these PAD values to return, but now those where it makes sense that they give an emotional feedback have been given PAD values.

P	A	D	Emotions
+	+	+	amused, excited, happy, ironic, interested, joking, proud, satisfied, self-confident, supportive
+	+	-	engaged, surprised
+	-	-	docile, thoughtful
+	-	+	certain, friendly
-	+	-	awkward, embarrassed, puzzled, shy, uncertain, uneasy
-	+	+	irritated
-	-	+	none
-	-	-	disappointed, hesitant, uncomfortable, unconfident, uninterested

Table 5.2: To better be able to understand where different emotions lay on the P, A and D scales this table was used.

To better be able to understand where different emotions lay on the P, A and D scales table 5.2 was used. It gives an idea of the values needed represent similar emotions and has been used as a reference point for emotions that were not listed in table 2.1. The new PAD values added to the behavioural presets can be found in table 5.3. Additionally, the emotions *hungry* and *sleepy* have been added to the emotion system in the controller using PAD values.

Emotions are hard to represent generally because of their subjective nature. What one person would classify as angry may be stronger or more dominant than another person's angry. It may take 'more' anger to reach what someone would call angry for some people than for other people. Because of this emotions have to be placed somewhere and then that emotion exists in a sphere around that point using the radius as the threshold. Also, during development, setting the initial values of an emotion may depend on a number of factors such as the developers mood or how familiar they are with a certain emotion.

To make the PAD axes more understandable I have tried to explain them using sentences. Arousal represents how exciting the feeling is, dominance how overpowering the feeling is, and pleasure represents how delightful it is. The most appropriate Norwegian translations I found to be *opphisselse*, *dominerende* and *gledelighet*, respectively.

One important point to remember is that the values in table 5.3 does not represent an emotion, but the change in emotion when the Cyborg is done executing the behaviour, which is then published on the */cyborg_controller/emotional_feedback* topic. The emotion system then uses the values it receives to change the current PAD values of the Cyborg. If those values were to represent specific emotions the variance in the values would be greater. Unfortunately, the new values have not been tested properly with the Cyborg. They will probably need some tuning to find the values that work best and to decide how strong the Cyborg's reaction should be. For example, what the emotional feedback to the `show_off` presets should be depends on what you want it to represent. It could be proud, confident or assertive, egocentric, mischievous, or happy. It all depends on what meaning you choose to give the behaviour.

Preset	**P**	**A**	**D**
R2D2 Short	0.05	0.10	0.02
Bored	-0.10	-0.10	-0.05
Transport Happy	0.10	0.10	0.10
Transport Neutral	0	0	0.01
Arrival	-0.05	-0.03	-0.05
Arrival, Elated	0.15	0.15	0.10
Arrival, Bored	-0.05	-0.40	-0.03
Arrival, Dignified	0.10	0.02	0.02
Arrival, Curious	0.02	0.40	-0.01
Arrival, Puzzled	-0.05	0.05	-0.40
Arrival, Angry	-0.20	0.20	0.15
Arrival, Unconcerned	-0.01	-0.04	0.01
Conveying Emotional State, Angry	-0.20	0.20	0.15
Conveying Emotional State, Curious	0.02	0.40	-0.01
Conveying Emotional State, Elated	0.15	0.15	0.10
Conveying Emotional State, Inhibited	-0.05	0	-0.04
Conveying Emotional State, Loved	0.30	0.20	-0.05
Conveying Emotional State, Unconcerned	-0.01	-0.04	0.01
Conveying Emotional State, Dignified	0.10	0.02	0.02
Conveying Emotional State, Bored	-0.05	-0.50	-0.03
Waking Up, Happy	0.10	0.10	0.10
Waking Up, Grumpy	-0.15	0.05	0.10
Show Off, MEA	0.20	0.10	0.15
Show Off, Music	0.15	0.10	0.10
Show Off	0.20	0.10	0.15
Navigation Go To, Music	0.15	0.20	0.05
Wandering Emotional	0	0.01	-0.01
Wandering Emotional, Happy	0.10	0.10	0.10
Sleepy	0.05	-0.15	-0.15
Exhausted	-0.05	-0.20	0.30
Speaking	0.10	-0.05	0.02
Make Way	0.05	0.10	0.20

Table 5.3: Some of the new PAD values for the behavioural presets.

Chapter 6

New Interpreters for the LED-dome

As part of the work with the behaviour module more interpreters have been developed. This is to give the Cyborg more to choose from so that it does not become too repetitive. Below, an explanation of the different new interpreters is given. For an explanation of how to add new interpreters see section 6.4.

A video cycling through the new additions can be seen on YouTube [24]. First shown is `suspension`, then `startup` and then the updated `eyes`. Unfortunately, the colours of startup became slightly distorted in the video so that the difference between some are hardly visible while others differ severely that in real life.

6.1 Update of Eyes

`Eyes` was the only static of the old interpreters, but as it was the most used, it needed more movement to give the Cyborg more life. The old eyes seemed to always be staring blankly into some void right above the head of any person in front of the Cyborg. To give the eyes more life they are now blinking. In listing 6.1 a somewhat simplified version of the code is shown. To make them blink, the function *random.randint(x,y)* is used. It sets the number of loops the render function should take before setting the eyes to closed for one loop, and then opening them again. This irregularity in the time it takes between each blink also adds to the life of the Cyborg. The function *rospy.Rate()* provides a way to keep a specific rate for a looping function and takes an int argument in Hz. Rate is used with *rospy.sleep()* which will raise a `rospy.ROSInterruptException` if node shutdown or similar occurs [13]. `Eyes` is run when the controller's state machine is in the idle state after receiving a message to start from the commander, and with various behaviours.

```python
#!/usr/bin/env python
import rospy
import system.settings as settings
import random

class Eyes():
    def __init__(self):
        self.rate = rospy.Rate(3)
        self.isStatic = False
        self.count = 0
        self.blink = random.randint(5,20)

    def render(self,input_data, output_data):
        if self.count == self.blink:
            # set the eyes closed

            self.count = 0
            self.blink = random.randint(5,20)
        else:
            # set the eyes open

        # set the edge of the LED-dome

        self.count += 1
        self.rate.sleep()
```

Listing 6.1: The updated version of eyes.

6.2 Race Stripes

The race stripes were created because of an idea to create a behaviour where the Cyborg would drive around in Glassgården pretending to be a race car. The Cyborg does not look very much like a car at all, but the idea came because a lot of race cars have some form of stripes. Either on the sides or over the roof and bonnet. In addition to the stripes, the Cyborg plays recordings of race cars on the track. The behaviour is fairly simple, the new parts only being the interpreter and the audio, the driving around is done by using *navigation_ wander*. The visual part is static as there is no need for a non-static because the main part is the movement and audio.

6.3 Startup and Suspension

When starting the Cyborg we need some kind of feedback to let us know it has started and is initialising. Then, when it is done initialising, it changes to a static or some other kind of animation to show that it is ready to continue. Like starting one of the modes like `Demo`, `Manual`, or the full system with behaviour. `Startup` is run by the behaviour launch file in the state startup. In this state, every node is initialised and started and it stays there until it receives a succeeded message. The state machine then moves to the state suspension in which the suspension interpreter is run.

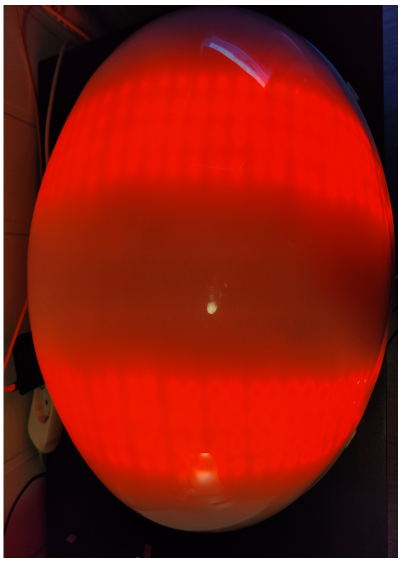


Figure 6.1: The interpreter visual `racing`.

The idea behind the startup interpreter was that it should visualise a form of something loading. There are innumerable forms and versions of those kinds of animations in the digital world. On MacBooks, the loading icon is a spinning colour wheel, which became the inspiration for the new startup interpreter. Because the LED-dome consists of 30 rows of led strips it was divided into 10 so that each colour uses 3 rows. To find the right colour the colour heel in fig. 6.2 was used and each hexadecimal value converted into RGB decimals. The wheel has 12 colours which complicated the looping somewhat, but a counter was used to keep track of the colours. Listing 6.2 shows startup's render function in which the two for loops iterate through the LED rows and the indexes of the LEDs to set every three rows to a colour. *Counter* keeps track of which colour is to be set for every three

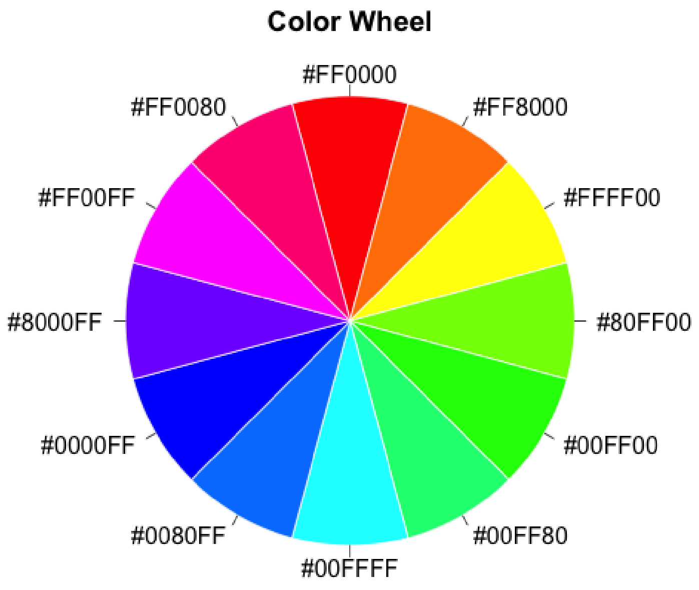


Figure 6.2: The basis for the colours used in the startup interpreter written in hexadecimals [42].

rows of LEDs. *Colours* keeps track of which colour is to be the first in the output array. For every time the output array is set, each colour is moved three rows.

To avoid draining the battery too fast, the LEDs' intensity has to be turned down. The intensity was halved and then tweaked from there. Red was unnecessary bright compared to the other colours and was turned down to [80,0,0]. The colour pink and magenta, FF0080 and FF00FF respectively, were hard to differentiate so the red in pink was turned up from 128 to 150. This made it appear more in between red and magenta. Blue, 0000FF, was turned turned down from [0,0,128] to [0,0,100], and green, 00FF00, was turned up from [0,128,0] to [0,150,0].

```python
def render(self, input_data, output_data):
    for i in range(0,settings.LED_ROWS-1, 3):
        for led in range(settings.LED_ARRAY_ROWS[i][1],
    settings.LED_ARRAY_ROWS[i+2][2]+1):
            if self.counter == 12:
                self.counter = 0

            output_data[led*3]   = self.colour_array[self.
    counter][0]
            output_data[led*3+1] = self.colour_array[self.
    counter][1]
            output_data[led*3+2] = self.colour_array[self.
    counter][2]
        self.counter += 1
    if self.colours == 12:
        self.colours = 0
    else:
        self.colours +=1

    self.counter = self.colours
    self.rate.sleep()
```

Listing 6.2: The code for the startup interpreter.

`Suspension` is a new static interpreter in which the Cyborg looks like it is sleeping. It is very similar to the interpreter `eyes`, but only the part where the eyes are closed is used. This is run when the controller's state machine is in the suspension state. The suspension state is a waiting state in which no mode has been chosen and sent to the Cyborg from the website. The Cyborg also enters this state if the current mode is aborted or sends a succeeded message.

6.4 How To Add New Interpreters

To add a new interpreter, add the new file to the folder `cyborg_led_dome/src/neural_interpreter/`. Preferably it should be called the same as both the class and the name of the interpreter to make it easier to keep track of. The class must have an `__init__` function to set any variables, and at least the *isStatic* variable used by *domecontrol.py* to determine if the LED-dome needs updating or not. If *isStatic* is false both the functions *rospy.Rate()* and *rospy.sleep()* should be included so that the loop takes the same time every time. The function actually setting the array of LED values has to be called `render` and take in *input_ data* and *output_ data*, which are the input from the MEA data and the array of LED values, respectively. Domecontrol also runs `sm.userdata.sm_interpreter.render( data,sm.userdata.sm_led_colors)` which is why render has to be called render.

Finally, the new interpreter has to be added in domecontrol in three different places in addition to importing the class from its file.

- Add the name of the interpreter to a new `elif` in the function *return_ interpreter()* and return the new class.
- Add the name to the function *update_ visualization_ mode()* in the part corresponding to the nonmea mode.
- Add the name to *set_ visualization_ mode_ callback().*

Chapter 7

Other Tasks

There is an abundance of smaller tasks in need of attention on the Cyborg. Be it a warning, an error or something that does not work precisely as intended. But they are not being worked on, either because one is implementing something enthralling and only want to work on that. Or because they are inadequate for a masters thesis. Simply because of the expectations and aim of such an undertaking, as well as a limited time frame. Regardless, they are things that have to be done when working on bigger projects. Software updates and documentation are often given lower priority and put aside because coding is more interesting. This was one of the points made in the author's specialisation projects [23]. Thus, how to go forward with updating the software used on the Cyborg has been studied and a setup script doubling as a list of dependencies and libraries used on the Cyborg has been created.

7.1 Updating ROS and Ubuntu

The Cyborg project is at the moment using Ubuntu 16.04 and ROS Kinetic Kame, but these will reach their end of life or standard support in April 2021. Because of this, some work was done to evaluate upgrading both. The current long term supported version of Ubuntu is 18.04, and the latest version of ROS is called Melodic Morenia. The Cyborg runs Xubuntu 16.04, which is a lighter version of Ubuntu 16.04.

Another thing to take into consideration is that Python 2.7 reached its end of support date on January 1. 2020. Which makes having the correct package versions for installing even more important. But note that Python 2 from the Ubuntu repositories will be supported until EOL for the Ubuntu releases [14].

When evaluating the possibilities and work needed to update Ubuntu and ROS, all current versions of packages in use were written down and included in the setup file. This is the file meant to be run when setting up a new computer to work on the Cyborg and can be found in chapter A. Many of them turned out to be the same or almost the same as the latest version of the packages. Those who have had a significant update did not always support Python 2.7 anymore. Because the new navigation stack is still being worked on and, at the time of examining the prospect of updating both ROS and Ubuntu, it had not been completely integrated with the Cyborg. Therefore its dependency package versions were not examined. The versions in use currently are however included in the setup script. Below, a list of some of the packages and current versions used on the Cyborg can be found.

Package	Current Version	Latest Version
libaria	2.9.4	-
libarnl	1.9.2a	-
arnl-base	1.9.2	-
mobilesim	0.9.8	-
pyttsx3	2.7	-
color	0.1.5	-
pandas	0.24.2	-
pyserial	3.0.1	3.4
numpy	1.16.5	1.16.6
VLC	3.0.7110	3.0.8

Libaria is installed using a .deb file which was downloaded before MobileRobots shut down. The same applies to *libarnl*, *arnl-base* and *mobilesim*. Most other packages are installed using `sudo apt install <package>` or `pip2 install <package>==<version>`. An update of both ROS and Ubuntu would mean going through all packages installed using the setup script and possibly other packages as well to find the right updated version. Most likely many of the open-source packages in use have already been updated to accommodate changes in ROS and Ubuntu. Unfortunately, there will be a need for changes in our packages. Python 2.7 can be installed on Ubuntu 20.04 which means that functions that no longer exist in Python 3 does not have to be replaced, but it might be a smart move nonetheless.

To be able to use Python 3 you either have to wait for the 2020 release of ROS1 called Noetic Ninjemys, or use ROS2 which has supported Python 3 from the beginning. It might be an idea to skip the ROS Melodic and jump straight to Noetic, as first making the Cyborg system compatible with the Melodic distribution and then Noetic will increase the workload considerably. There are ways to make Python 3 compatible with the older distributions of ROS and this would require building Python 3 from source, which in the author's opinion is unnecessary and it would be better to upgrade the whole system for access to Python 3.

There are also ways of using both ROS1 and ROS2 together on the same system, which requires a bridge between the two. For example the package `ros1_bridge`. More information on that can be found on the ROS wiki and forum pages. This is especially relevant for the object detection and computer vision use-cases as ROS2 provides software for object detection which cannot be found in ROS1. Because object detection is being worked on, but not integrated on the Cyborg yet, upgrading to ROS2 is not a priority. Migrating entirely to ROS2 would mean an overhaul of the rest of the software on the Cyborg as well.

When testing on Ubuntu 18.04, the system did build and launch, but running the system created an abundance of errors. One problem which emerged was the connection with the LED-dome seemed to be failing. It seemed to receive messages on topics and forward them to the LED-controller correctly, but the LEDs did not turn on. No changes to the LED-controller had been done and when connecting it to the old system it worked as before. It was therefore concluded that the updates were the cause. Other odd behaviour included some nodes which either did not publish messages or functions which did not return the correct output. Unfortunately, it would have taken too long to figure out the causes of all these errors and then correct them, and further work on this was therefore abandoned.

Updating the system will take a lot of work and would either require a person committing their specialisation project or thesis to update the system, or multiple students working on different parts to update their respective parts. I cannot guarantee that all packages needed are included in the setup script, but most often packages that are not installed will yield 'package not found' errors which are relatively easy to work with.

For now, there is no need to update just because Python 2 is not supported any more, neither is it crucial to update the current Ubuntu or ROS distributions. As long as older releases of packages in use on the Cyborg are accessible to install on new computers they will not pose a problem in that sense. The main reason to update would be to have access to new packages that might not support older releases. Older versions of the software will not disappear, but it becomes increasingly important to include version numbers in all dependencies and the setup script.

7.2 Setup Script

Last autumn, when starting the work on the specialisation project, one recurring problem was that packages the Cyborg software depended on were not listed anywhere. That meant that the most efficient way of discovering which dependencies were needed was to run the code and see what errors appeared. Usually, they were 'package not defined' or 'no module named' errors, but sometimes they were more complex and needed more work to find the correct version of a package or the correct package.

This, along with checking all imports, became the most efficient way of finding any more dependencies which had been installed but were not written down when testing, when working on creating a setup script as complete as possible. When creating new modules for the Cyborg, you try to take note of any packages needed, especially those which have to be installed. Unfortunately, they can easily become hard to keep track of.

There existed a setup script, but it did not seem to be up to date. It did not run properly and was missing dependencies for entire nodes. This script was also run as root which gave some peculiar behaviour when trying to run the Cyborg system as those commands seemed to have to be run as root too. Which should not be done. Thus, any commands that do not require `sudo` when run will not have it unnecessarily added by running the whole script as root.

An opportunity presented itself when NTNU closed and everyone had to work from home because of the Coronavirus. As we were not allowed to bring any of NTNU's computers home, work had to be done from private laptops or desktops. Because the author only has one laptop with Ubuntu 18.04 and this was not able to run the Cyborg software, the other options were a dual boot with Ubuntu 16.04, installing Ubuntu 16.04 on a virtual machine, or accessing the desktop in the office at NTNU using ssh, TeamViewer or similar.

At first, TeamViewer was used but this ended up being slow because of the WiFi connection. It then was decided to use a virtual machine, which would prove to be a poor decision because of the terrible performance of the virtual machine. With only 4 GB RAM on the laptop versus 32 GB on the desktop, this proved to be amazingly slow. Building the software could take over an hour. Fortunately, Python is not a compiled language so any smaller changes did not generate a need to build again.

But, because of the newly installed Ubuntu virtual machine, none of the dependencies of the Cyborg software had been previously installed and would show up as some kind of error. Thus the first parts of the setup script installing ROS and cloning the Cyborg repository, found in chapter A, was tested and any errors fixed. Then testing was done to find all modules missing. The most important part of this is that because the Cyborg uses relatively old packages the correct version has to be included for as many as possible of the installs. For any packages installed using `sudo apt-get install ros-kinetic-<package>` the version is already included because of the *kinetic* part, and thus all those will work with the Cyborg software. Any packages installed using pip has to be compatible with Python 2.7. Most have had several updates since they were first included on the Cyborg and most only support Python 3 as Python 2.7 has reached its EOL.

For the new parts with behaviour trees most dependencies were already noted in *setup.py* files, except for the installation of the `behavior3py` library. The setup files are read by catkin when running the command `catkin_make` and thus, does not have to be run by the setup script. *Behavior3Editor* contained mainly two files used to install any missing dependencies called *package.json* and *bower.json*. They had to be edited to fit the Cyborg as certain versions or scripts were missing. *Nodejs* version 10.x also had to be installed. Any newer versions did not support Python 2.7 and could not be used. When installing *behavior3py* one issue quickly arose, the library only existed as source code on GitHub and had to be built from source. As this was new to the author, different methods for building from source were researched and tested. Because this is someone else's library, creating a package and publishing it so others can install it using pip or similar felt unnecessary. Therefore it was decided to use a simpler solution in which the library is built and installed locally using a straightforward setup file.

The new navigation stack was merged into the master branch and has been included in the search for missing dependencies, and it proved to be missing quite a lot. All of which have been added to the setup script. The same applies to the new website and its nodes. Any extra files needed for the installs have been added to its folder in the setup directory in the Cyborg workspace. This applies to the *behavior3py* library and the *.deb* files needed to install software from MobileRobots which is no longer readily available. The script has been tested, and though it takes some time, it worked well. A few pauses have been added to the script so that any errors that occur can be found and corrected.

In addition to the setup script, a requirements file has been added. This is to make it even easier to install the correct versions of the required Python libraries. This is created using the command `pip freeze > requirements.txt`. This showed that of the libraries used, around 10 had some form of weakness like security. This adds to the reasons why the Cyborg software should be updated in the future.

A script running the `roslaunch` command at startup of the Cyborg has also been created. A small issue presented itself because the Cyborg's Xubuntu uses the xfce4-terminal, while Ubuntu uses gnome-terminal. This means that there are two different commands in the script, one for each OS. The reason to have this script start a terminal window is so that it is possible to access the Cyborg and check the output in the terminal. Because this is seldom used on the development desktops their respective command is commented out. Both versions of the script have been tested and worked great.

7.3 Audio

To give the behaviour module greater flexibility and developers more to choose from in the future, an abundance of new audio files have been added to the audio node. These range from short expressions like saying yes or no, to longer quotes from movies or music that can be looped. While the complete list can be found in chapter B, some of them include quotes like "Here we go", "I am you father", "You talkin' to me?", "Where is my super suit?", "Just a flesh wound" and "Tis but a scratch". Most of them are intended for use when meeting and interacting with people in Glassgården. For example, if someone tries to pass the Cyborg it may play the *none shall pass* audio file. Or it can play "Just a flesh wound" or "Tis but a scratch" if it bumps into something. Other examples can be when meeting a new person it could play "Hello there", "Got a minute?" or "Do you know who I am?". The same file could also be used for a number of different situations or behaviours. The file *star_ wars_ cantina_ band* has been added to the `suspension` mode which seemed fitting. *here_ we_ go* as been added to be played when it starts the state *idle*, but the audio file *showtime* could very well have been used too. For the new *racing* behaviour the file *racing_ford_ gt40* has been as the playback in its behavioural preset. The music from Monty Python's sketch The Black Knight was added to one of the wandering presets, while another has received the BB-8 talking audio.

Chapter 8

Discussion

One part of this thesis has been to explore the use of behaviour trees for use on the Cyborg. The other part has consisted of a variety of tasks which, while not as major on their own, were needed to bring the Cyborg closer to its goals. At the beginning of this work, the Cyborg was reevaluated and parts of the Cyborg which needed work were identified. Among these were the continuation of the work on the new navigation stack and the new GUI website, prompting a need to reorganise the software structure of the Cyborg somewhat. And because the Cyborg's software dependencies are several years old, an update of that software.

Structure

The Mode Selector Box had stopped working last year and because the new website was already in development, it was decided not to fix it at this time. Creating a new version of the Mode Selector Box may not be a bad idea, as you would have something to at least change mode or stop the Cyborg, giving it some redundancy. To interact with the rest of the Cyborg's software a new node called `commander` was created which received and transmits messages to and from the website. The Cyborg's structure is now more clear and it has been prepared for implementing the modes `Demo` and `Manual` as well as any additional modes in the future.

Update ROS and Ubuntu

After Python 2.7 reached its end of life in January and both ROS Kinetic and Ubuntu 16.04 reaching their end of life next year, it was decided that upgrading would be the right move. But, upgrading not just the OS and ROS but all the nodes to be compatible with them and not use Python 2.7 will be quite an undertaking. It has to be taken into account that not all open-source nodes from ROS are updated to be compatible the instance a new distribution of ROS is released because they are developed by users, and some may not be updated at all. An additional point is that updating the Cyborg every time a new distribution of ROS and Ubuntu is released may be almost as much work as abstaining from updating every time, and rather update every other release or more seldom than that. If it were the same team working on the Cyborg continuously it might require less effort to go for the first option rather than the latter. Continuing to use Ubuntu 16.04 and ROS Kinetic can create problems in the future if there are nodes or functionality that are not compatible with older software or are only created for the, at that time, current releases. While none of the current libraries and software probably will not disappear, they might be removed from their servers which will make them more difficult to obtain and install.

Setup Script

Both working as documentation and a way to install all dependencies on a new system, the setup script script has been completed and now includes all libraries needed to work on the Cyborg. Though it was tested and worked at the time, does not mean that it is done. Every time a new node is added or something is changed in an existing node, the script has to be updated. Otherwise it looses its purpose. The main way to keep it up to date is to add or remove anything as soon as nodes are modified. Another way is to try to run it on a "clean" system. That way it will issue errors or warnings for missing modules or installs.

PAD

The application of emotional feedback from different behavioural presets were missing, and thus all presets where it makes sense that they give emotional feedback have received PAD values. They should be tested more thoroughly, but this was not possible at the time. Previous years it have been suggested to create some form of emotion configuration guide, which is a good idea. A problem with this though, is that every person will most likely assign different PAD values with the same emotion. What PAD values they set may depend on a number of different factors like how they are feeling that specific day. There is no definitive answer to which values should correspond to each emotion. Which is why a threshold, or a radius around every value, is used in the emotional module. But a guide with some directions, beyond existing values like in table 2.1 and where different emotions lie on the scale like in table 5.2, would probably be a good idea.

Behaviour Module

The use of an FSM in the main controller creates limitations to the Cyborg's behaviour and the readability of its code. The behaviour of the Cyborg will mainly be unstructured which finite state machines do relatively poorly, but behaviour trees handle unstructured behaviour well. They were therefore researched and their possibilities on the Cyborg evaluated.

A high-level tree was made as a way to begin implementing the controller. Previously only very specific subtrees had been created. Some design choices have to be made going forward. Specifically a choice has to be made regarding how much of the existing ROS nodes should be kept as is and use communication channels between the different nodes, or if functionality from the ROS nodes should be moved into the behaviour tree nodes. Using the existing ROS nodes as behaviour tree nodes might not work very well because they are quite large and complex. Behaviour tree nodes only contain functionality necessary to execute whatever it is it should do, like the node `move_to`. It is an example that, with the current navigation module, only needs to publish a message on the correct topic and let the navigation module handle the execution. Unfortunately, the previously existing work with behaviour trees used `ros-arnl` and replacing it with the new navigation stack was not prioritised after the new stack was completed. Behaviour trees can be combined with several state machines, meaning there is no need to completely remove all state machines abruptly.

Three different behaviour tree libraries were compared to the library previously used and it was concluded that if the decision to change the library used on the Cyborg, the preferred option would either be *BehaviorTree.CPP* or *py_trees*. This is because they both include features that were deemed important for the Cyborg. These features include a GUI and a way to visualise the tree during runtime, that they be written in either Python or C++, they provide functionality for parallel execution of nodes and a blackboard. They are also well documented and currently maintained.

While the concept of behaviour trees are fairly easy to understand, it would probably be advantageous if future student wanting expand the use of behaviour module has previous experience with behaviour trees. They would then know of the limitations and full potential of behaviour trees.

8.1 Proposed Tasks for Future Work

Presented below is a list of proposed tasks that are needed to bring the Cyborg to its desired state and tasks that were either not completed or need re-implementation.

Text-to-Speech - A decision has to made whether to keep this module or discard it. As of now, it uses a Python library called `espeak` which works but sounds very mechanical. If the decision to keep and improve it, some recommendations for its replacement include IBM's Watson or Google Cloud text-to-speech. They are both online, which would free up processing power from the Cyborg.

Giving the Cyborg moods - The proposal is to use the MEA data to create moods represented using PAD values. Other sources that could influence the mood could be the amount of battery charge left or the current weather.

More active usage of MEA data - There are many possibilities with the MEA data, and two ideas which have not been realised yet are to use it to create feelings and use that to choose behaviours, or use the data to set the emotional feedback when the Cyborg is done executing behaviours. They could also be used in a feedback loop either live to the cells at St. Olav's or from a file.

Behaviour Trees - Use behaviour trees to create a more complex controller and give the Cyborg more lifelike behaviour.

Docking Behaviour - The Pioneer robot was delivered with a charging station and automatic docking, but with the new navigation stack this is no longer functional.

Recovery Behaviour - The Cyborg would need some kind of recovery programme if it, for example, gets stuck or loses track of its position. It should be able to do this without human intervention or at least let a human know that it needs assistance.

Demo & Manual modes - These modes have been added to the current state machine but lack implementation. `Manual` is intended to utilise the built-in possibilities to control the Cyborg. Either from the new website or with a joystick which came with the Pioneer robot. `Demo` is intended for use at a stand or a demonstration where the Cyborg either lacks a map to navigate by or when it is standing still. It should be able to play audio and show off visualisations on the LED-dome.

Object Detection - Use the newly created implementation using object detection on the Cyborg. Object detection comes with endless possibilities for the behaviours and navigation, either to trigger a behavioural preset or to navigate by and to be used for recovery.

Chapter 9

Conclusion

The amount of work needed to upgrade the Cyborg to the latest software, what those upgrades would be, and whether those upgrades are relevant to the Cyborg have been evaluated. To continue to use ROS1 was concluded to be the best action going forward, and wait to upgrade to ROS2. Upgrading to Ubuntu 20.04 and ROS Noetic Ninjemys is recommended, though this will take a considerate amount of work.

A setup script for the Cyborg has been created and doubles as documentation of which libraries are used and installs are needed on the Cyborg. This has been tested and worked as expected. Additional visual interpreters have been created for the LED-dome and other interpreters have been modified to give the Cyborg more combinations of behavioural presets to choose from. The number of behavioural presets have been expanded and the majority have received emotional feedback PAD values.

The implementation of the behaviour module using behaviour trees is not fully ready to replace the existing FSM controller at this point. Both are therefore still on the Cyborg, but the behaviour trees have their own branch on GitHub so that any student who would like to continue work on the behaviour module using behaviour trees have it available. The FSM implementation is the current implementation of the master branch. Four different behaviour tree libraries were compared and it was concluded that if the decision to change the library used on the Cyborg, the preferred option would either be *BehaviorTree.CPP* or *py_trees*. Colours can be used in the interpreters for the LED-dome to awake specific feelings and emotions in people. This effect was studied and the work presented can be used as a reference work for future implementations of interpreters and behaviours.

The goal of this thesis has not been to criticise previous architectural choices and implementations and point out imperfections, but to delve into a field that has received much attention in the last years and study how it can be used on the Cyborg. Behaviour trees provide functionality the Cyborg can employ to give it a more lifelike appearance and behaviour.

Appendices

Appendix A

Setup Script

```
#!/bin/bash

echo "Setup script running..."
echo "WARN - Specific versions of packages are needed in this project and when installing warnings about deprecated or old software may show up. This is expected."
echo "This script was tested and worked in may 2020, with new updates to libraries or changes to any of the nodes this may not be the case any longer."
read

# All commands should also be updated to include versions.

echo "---------- General installs and setup ----------"
echo "to continue press enter"
read

sudo apt-get update
sudo apt-get install git
sudo apt install python2.7
sudo apt install python-pip
pip install --upgrade pip==20.0.2


echo "---------- Install ROS ----------"
echo "to continue press enter"
read

```

```bash
## ROS
# Setup your computer to accept software from packages.ros.org:
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
# Set up your keys
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654

sudo apt-get update
# Full ROS installation
sudo apt install ros-kinetic-desktop-full
# Find avaliable packages
apt-cache search ros-kinetic
read -p "Available packages (above). Take a note if some are needed. Press ENTER to continue."
# Initialise rosdep
sudo rosdep init
rosdep update
# Setup the environment
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
# Dependencies for building packages
sudo apt install python-rosinstall python-rosinstall-generator python-wstool build-essential
# Create a workspace
mkdir -p ~/catkin_ws/


echo "---------- Clone Cyborg repo from Git ----------"
echo "to continue press enter"
read

## Clone Cyborg Repo from git, and place it in the right directory
cd ~/catkin_ws
git clone https://github.com/thentnucyborg/CyborgRobot.git  # clones the master branch

cd ./CyborgRobot
echo "Use the master branch? Please type the branch name or 'yes' if you wish to use the master branch."
```

```bash
read branchname
if [ "$branchname" == "yes" ]
then
  echo "Using the master branch"
else
  echo "Using the "$branchname" branch"
  git checkout $branchname
fi

mv ~/catkin_ws/CyborgRobot/* ~/catkin_ws/ #move all files and folders to the workspace
mv ~/catkin_ws/CyborgRobot/.* ~/catkin_ws/  #move all hidden files and folders to the workspace. Ingore the message saying . and .. cannot be moved
echo "Please ignore the message saying . and .. cannot be moved"
rm -rf ~/catkin_ws/CyborgRobot  #delete the now empty folder



echo "---------- Install Arnl and ARIA .debs ----------"
echo "to continue press enter"
read

cd ~/catkin_ws/setup/installs
sudo dpkg -i arnl-base_1.9.2+ubuntu16_amd64.deb
sudo dpkg -i libarnl_1.9.2a+ubuntu16_amd64.deb
sudo dpkg -i libaria_2.9.4+ubuntu16_amd64.deb
sudo dpkg -i mobilesim_0.9.8+ubuntu16_amd64.deb


echo "---------- Install GUI ----------"
echo "to continue press enter"
read

pip2 install pymongo==3.10.1
pip2 install pymongo[srv]
sudo apt-get install ros-kinetic-rosauth
sudo apt-get install ros-kinetic-rosbridge-server
sudo apt-get install ros-kinetic-rospy-message-converter
sudo apt-get install rosbash
```

```
echo "---------- Installs for New Navigation ----------"
echo "to continue press enter"
read

## Installs for Navigation stack
cd ~/catkin_ws/src
sudo apt-get install ros-kinetic-navigation
sudo apt-get install ros-kinetic-tf2-sensor-msgs
sudo apt-get install ros-kinetic-tf2-geometry-msgs
sudo apt-get install ros-kinetic-cmake-modules
sudo apt-get install ros-kinetic-tf2-kdl
sudo apt-get install ros-kinetic-kdl-parser
sudo apt-get install ros-kinetic-move-base
sudo apt-get install ros-kinetic-image-transport
sudo apt-get install ros-kinetic-interactive-markers
sudo apt-get install ros-kinetic-python-qt-binding
sudo apt-get install ros-kinetic-resource-retriever
sudo apt-get install libogre-1.9-dev
sudo apt-get install libsdl-dev
sudo apt-get install libsdl-image1.2-dev
sudo apt-get install libbullet-dev
sudo apt-get install libassimp-dev assimp-utils

sudo apt-get install openni2-doc && openni2-utils && openni-
    doc && openni-utils
sudo apt-get install libopenni0 libopenni-sensor-pointclouds0
sudo apt-get install libopenni2-0
sudo apt-get install libopenni-sensor-pointclouds-dev
sudo apt-get install libopenni2-dev
sudo apt-get install libopenni-dev
sudo ln -s /usr/lib/python2.7/dist-packages/vtk/
    libvtkRenderingPythonTkWidgets.x86_64-linux-gnu.so /usr/
    lib/x86_64-linux-gnu/libvtkRenderingPythonTkWidgets.so
sudo update-alternatives --install /usr/bin/vtk vtk /usr/bin/
    vtk6 10

git clone https://github.com/ros-visualization/rviz.git -b
    kinetic-devel
git clone https://github.com/ros-planning/navigation.git -b
    kinetic-devel


echo "---------- Install SMACH ----------"
echo "to continue press enter"
read

## Install SMACH
sudo apt-get install ros-kinetic-executive-smach
sudo apt-get install ros-kinetic-executive-smach-
    visualization
sudo apt-get install python-pyqt5
sudo apt-get install python-qt-binding


echo "---------- Install Aduio ----------"
echo "to continue press enter"
read

## Install for Audio node
pip2 install -Iv pyttsx3==2.7 #-I ignores installed packages,
     -v prints/verbose
pip2 install python-vlc==3.0.7110


echo "---------- Install Command node ----------"
echo "to continue press enter"
read

## Install for Command node
pip2 install npyscreen


echo "---------- Install Controller ----------"
echo "to continue press enter"
read

## Install for Controller node
sudo apt install graphviz-dev
pip2 install pygraphviz
#Alternatively, run: sudo apt-get install python-pygraphviz


echo "---------- Install LED Dome node ----------"
echo "to continue press enter"
read

```

```
## Install for Led Dome node
pip2 install colour==0.1.5
pip2 install numpy==1.16.6
pip2 install pandas==0.24.2
pip2 install pyserial==3.0.1
pip2 install pyopengl
pip2 install pyopengl-accelerate
pip2 install pytz


echo "---------- Install Behaviour Trees ----------"
echo "to continue press enter"
read

## Behavior Trees
pip2 install networkx==2.2
chmod +x  ~/catkin_ws/src/rqt_behavior_tree/scripts/rqt_bt
# install nodejs for behavior3editor
curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash
    -
sudo apt-get install -y nodejs    # also installs npm
#install bower
sudo npm install -g bower
# install dependencies for behavior3editor
cd ~/catkin_ws/src/behavior3editor
sudo chown -R $USER:$GROUP ~/.npm
sudo chown -R $USER:$GROUP ~/.config
npm install
bower install
sudo npm install --global gulp@3.9.1
sudo apt-get install python-scipy
# install b3 module
cd ~/catkin_ws/setup/installs/behavior3py
sudo python setup.py install



echo "---------- Setup UDEV Rules ----------"
echo "to continue press enter"
read

## Set up UDEV rules
sudo cp ~/catkin_ws/setup/90_cyborg_usb_rules.rules /etc/udev
```

```
    /rules.d/
sudo udevadm control --reload
sudo udevadm trigger


echo "---------- other ----------"
echo "to continue press enter"
read

## Other
sudo apt-get install sqlitebrowser  #tool for editing
    databases
pip install -r requirements.txt

# Make  python  and  bash  scripts  executable
find ~/catkin_ws/src/ -name '*.py' -exec  chmod +x {} \;
find ~/catkin_ws/src/ -name '*.sh' -exec  chmod +x {} \;


echo "---------- Finish setting up catkin_ws ----------"
echo "to continue press enter"
read

# Finish setting up the workspace
source ~/.bashrc
source /opt/ros/kinetic/setup.bash
cd ~/catkin_ws
catkin_make
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc #
    Opening new terminal runs source command, so we dont have
    to source workspace each time.
source devel/setup.bash


## Base requirements
sudo usermod -a -G dialout $USER  #add user to dialout group
sudo apt autoremove



echo "Setup script ended..."
echo "---------------------"

```

```
# Relogin is required for last cmd to take effect
echo "You must logout and back in for userpivileges to take effect..."
echo "You might want to change a value in /usr/local/Aria/params/pioneer-lx.p to flip the sensor output on the Cyborg the right way up. See cyborg_navigation/README.md for how to."
echo "If changes to the code on the led-controller are needed (the NodeMCU ESP32), follow the install instructions in cyborg_ros_led_dome/README.md"
```

Listing A.1: Setup script for the NTNU Cyborg. The `read` commands are added create a pause to be able to check for any errors or warnings that may appear.

Appendix B

Audio Files

Here is a list of the new audio files added to the Cyborg, and some examples of usage.

- **BB8** - Wandering
- **Darth Vader breathing** - Wandering
- **Do you know who I am?** - Interacting with people
- **English motherfucker, do you speak it?**
- **I fart in you general direction**
- **Just a flesh wound** - If it bumps into something
- **Ford model T startup** - Starting to move or similar
- **Get to the choppa** - Moving away from something
- **Got a minute** - Meeting someone
- **Hamster and elderberry**
- **Hello there** - Meeting someone
- **Here we go** - Starting something or changing behaviour
- **Here's my card** - Leaving someone
- **Hey** - Meeting someone

- **I am your father** - During interactions
- **I'll be back** - Leaving someone or moving away from them
- **It's a trap**
- **Mind your own business** - Interacting with people
- **Monty Python - The Black Knight** - Music that can be played in different situations
- **No**
- **No problemo** - During interactions
- **Nobody expects the spanish inquisition**
- **None shall pass**
- **R2D2 - Dagoba** - Wandering
- **R2D2 - Weeeooooow** - Typical R2D2 behaviour
- **Racing** - Wandering or racing behaviour
- **Racing - Ford GT40** - Wandering or racing behaviour
- **Say what again** - During interactions
- **Scream** - When people round a corner and meet the Cyborg, or
- **Showtime** - Starting something or changing behaviour
- **Star Wars - Cantina band** - Wandering or when in idle
- **Star Wars - Jump to lightspeed**
- **Taunt you a second time**
- **The power of the dark side** - During interactions
- **The public is in danger** - Followup to the super-suit audio
- **Tis but a scratch** - If it bumps into something

- **Uh oh**
- **Victory is mine**
- **Wall-E** - Saying hi to someone
- **Well, hello** - Meeting someone
- **What a strange person** - During interactions or when leaving someone
- **Where is my super suit?** - Wandering
- **Why so serious?** - During interactions
- **Wow 1**
- **Wow 2**
- **Yes**
- **You talkin' to me?** - During interactions or meeting someone

References

[1] EiT Group 3. *Project Report - Design and Production of the LED-dome for the NTNU-Cyborg.* Tech. rep. EiT Village - Kyborg, NTNU, 2018 (cit. on p. 11).

[2] Mariusz Szwoch Agata Kolakowska Agnieszka Landowska, Wioleta Szwoch, and Michal Wróbel. *Modelling Emotions for Affect-Aware Applications, in Information Systems Development and Applications.* Available at http://www.wzr.ug.edu.pl/nauka/upload/files/Information%20systems%20development%20and%20applications.pdf, pp.55-69. Faculty of Management, University of Gdańsk, 2015. ISBN: 978-83-64669-06-4 (cit. on p. 27).

[3] Thomas Rostrup Andersen. "Controller Module for the NTNU Cyborg". MA thesis. NTNU, 2017 (cit. on pp. 6, 42, 43).

[4] Areg Babayan. *Consolidating and Visualizing Biological Neural Network Activity on the NTNU Cyborg.* Tech. rep. Department of Engineering Cybernetics, NTNU, 2018 (cit. on pp. 5, 13).

[5] Areg Babayan. "Cyborg 3.0". MA thesis. NTNU, 2019 (cit. on pp. 5, 11, 37, 43–45).

[6] *BehaviorTree.CPP.* URL: https://github.com/BehaviorTree/BehaviorTree.CPP (cit. on p. 53).

[7] Ole Martin Brokstad. "Object Detection for the NTNU Cyborg". Final title not available at the time. MA thesis. NTNU, 2020 (cit. on p. 5).

[8] *Coding Standard, Testing and Style Guide - Naming Conventions.* URL: https://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html (cit. on p. 14).

[9] Michele Colledanchise. *BT++ A ROS Behaviour Tree Library in C++, User Manual.* URL: https://github.com/miccol/ROS-Behavior-Tree/blob/master/BTUserManual.pdf.

[10] Michele Colledanchise. *ROS-Behavior-Tree.* URL: https://github.com/miccol/ROS-Behavior-Tree (cit. on p. 53).

[11] Michele Colledanchise. *ROS-Behavior-Trees.* URL: http://wiki.ros.org/behavior_tree (cit. on p. 46).

[12] Michele Colledanchise and Petter Ögren. "Behavior Trees in Robotics and AI: An Introduction". In: *CoRR* abs/1709.00084 (2020). arXiv: 1709.00084. URL: http://arxiv.org/abs/1709.00084 (cit. on p. 32).

[13] Ken Conley. *Rospy Time.* URL: http://wiki.ros.org/rospy/Overview/Time (cit. on p. 70).

[14] Kyle Fazzari. *Some things to know as Python 2 approaches EOL.* URL: https://discourse.ros.org/t/some-things-to-know-as-python-2-approaches-eol/11175 (cit. on p. 76).

[15] *FreeRTOS API Reference.* URL: http://web.ist.utl.pt/~ist11993/FRTOS-API/index.html (cit. on p. 14).

[16] Patrick Goebel. *Pi Trees.* URL: http://wiki.ros.org/pi_trees (cit. on p. 46).

[17] Lasse Göncz. "New Navigation Stack for the NTNU Cyborg". Final title not available at the time. MA thesis. NTNU, 2020 (cit. on pp. 4, 15).

[18] Lasse Göncz. *Reimplementing the Navigation Stack on the NTNU Cyborg in a Simulated Environment.* Tech. rep. Department of Engineering Cybernetics, NTNU, 2019 (cit. on p. 15).

[19] Donald F. Hoffman and Shannon B Cuykendall. *From Color to Emotion, Ideas and Explorations.* Tech. rep. Cognitive Sciences, University of California, Irvine, 2008. URL: http://www.cogsci.uci.edu/~ddhoff/FromColorToEmotion.pdf (cit. on p. 59).

[20] Holistic3d. *Introduction to Behaviour Trees.* URL: https://en.wikipedia.org/wiki/BibTeX.

[21] *Installing the ESP32 Board in Arduino IDE.* URL: https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-mac-and-linux-instructions/ (cit. on p. 12).

[22] Matteo Iovino et al. *A Survey of Behavior Trees in Robotics and AI.* 2020. arXiv: 2005.05842 [cs.RO]. URL: https://arxiv.org/abs/2005.05842 (cit. on p. 52).

[23] Johanne Døvle Kalland. *Exploring Visualisations and Behaviour.* Tech. rep. Department of Engineering Cybernetics, NTNU, 2019 (cit. on pp. iii, 9, 75).

[24] Johanne Døvle Kalland. *New animations for the NTNU Cyborg.* URL: https://youtu.be/wJyiJOJFOjQ (cit. on p. 69).

[25] Steinar Kragerud. "NTNU Cyborg with Communicational Abilities". MA thesis. NTNU, 2016 (cit. on p. 6).

[26] Brent Lance and Stacy Marsella. "Glances, glares, and glowering: How should a virtual human express emotion through gaze?" In: *Autonomous Agents and Multi-Agent Systems* 20 (May 2010), pp. 50–69. DOI: 10.1007/s10458-009-9097-6.

[27] *License Details FreeRTOS.* URL: https://www.freertos.org/a00114.html (cit. on p. 13).

[28] MediaLabs. *Color Psychology: How color influence our mind.* URL: http://blog.medialabs.in/2015/12/17/color-psychology-how-colors-influence-our-mind/ (cit. on p. 58).

[29] Albert Mehrabian. *General and Precise Tests of Emotions, Feelings or Affect.* URL: http://www.kaaj.com/psych/scales/emotion.html#definition (cit. on p. 28).

[30] Morten Mjelva. "Control System and Object Detection System for the NTNU Cyborg". MA thesis. NTNU, 2018 (cit. on pp. 6, 37, 46).

[31] Adept MobileRobots. *Pioneer LX Mobile Research Platform.* 2013 (cit. on p. 9).

[32] Adept MobileRobots. *Pioneer LX, User's Guide.* 2013 (cit. on p. 9).

[33] Costanza Navarretta. “Predicting emotions in facial expressions from the annotations in naturally occurring first encounters”. In: *Knowledge-Based Systems* 71 (Nov. 2014), pp. 34–40. DOI: 10.1016/j.knosys.2014.04.034.

[34] Casper Nilsen. “GUI and Website for the NTNU Cyborg”. Final title not available at the time. MA thesis. NTNU, 2020 (cit. on p. 5).

[35] NodeMCU. *NodeMCU ESP-32S Documentation.* URL: https://nodemcu.readthedocs.io/en/dev-esp32/ (cit. on p. 12).

[36] *NodeMCU-32S Lua Module.* URL: https://www.smart-prototyping.com/NodeMCU-32S-Lua-WiFi-ESP32-module (cit. on p. 13).

[37] Jason M. O’Kane. *A Gentle Introduction to ROS.* Available at http://www.cse.sc.edu/~jokane/agitr/. Independently published, Oct. 2013. ISBN: 978-1492143239 (cit. on p. 24).

[38] Renato de Pontes Pereira and Paulo Martins Engel. *Behavior3py.* URL: https://github.com/behavior3/behavior3py (cit. on pp. 46, 53).

[39] *ROS Actionlib Documentation.* URL: http://wiki.ros.org/actionlib (cit. on p. 25).

[40] *ROS Documentation.* URL: http://wiki.ros.org/ (cit. on p. 17).

[41] Sam. *Setting up the Pioneer LX Mobile Base and running the demos.* URL: https://afsyaw.wordpress.com/2018/02/28/setting-up-the-pioneer-lx-mobile-base-and-running-the-demos/ (cit. on p. 9).

[42] Gaston Sanchez. *Colortools: Color Wheel.* URL: https://gastonsanchez.wordpress.com/2012/08/31/colortools-color-wheel/ (cit. on p. 72).

[43] Chris Simpson. *Behaviour Trees for AI: How They Work.* URL: https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php.

[44] *SMACH Package Summary.* URL: http://wiki.ros.org/smach (cit. on p. 34).

[45] splintered-reality. *py_trees.* URL: https://github.com/splintered-reality/py_trees (cit. on p. 53).

[46] Espessif Systems. *ESP-WROOM-32.* URL: https://www.espressif.com/en/products/hardware/esp-wroom-32/overview (cit. on p. 12).

[47] Multichannel Systems. *MEA2100-Systems*. URL: https://www.multichannelsystems.com/products/mea2100-systems#overview (cit. on p. 12).

[48] *The Cyborg Wiki*. URL: https://www.ntnu.no/wiki/display/cyborg/ (cit. on pp. 16, 18).

[49] thentnucyborg. *CyborgRobot Github*. URL: https://github.com/thentnucyborg/CyborgRobot (cit. on p. 13).

[50] Dirk Thomas. *rqt_graph*. URL: http://wiki.ros.org/rqt_graph (cit. on p. 46).

[51] Jørgen Waløen. “The NTNU Cyborg v2.0: The Presentable Cyborg”. MA thesis. NTNU, 2017 (cit. on pp. 6, 11).

Johanne Døvle Kalland

The NTNU Cyborg - Behaviour Module & Behaviour Trees

NTNU
Norwegian University of
Science and Technology