NTNU
Norwegian University of
Science and Technology

Controller Module for the NTNU Cyborg

Thomas Rostrup Andersen

Master of Science in Cybernetics and Robotics
Submission date: February 2017
Supervisor: Sverre Hendseth, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

NTNU – Trondheim
Norwegian University of
Science and Technology

DEPARTMENT OF CYBERNETICS AND ROBOTICS

Controller Module for the NTNU Cyborg

Written by

Thomas Rostrup Andersen

Date: February 26, 2017

Supervisor: Sverre Hendseth

Abstract

The work with this master thesis was carried out as a part of the NTNU Cyborg project at the Norwegian University of Science and Technology (NTNU). Several modules have already been developed for the Cyborg. These modules include a variety of functionality which may cause conflict if they run at the same time. The modules detect events and act on them. A controller module is implemented for controlling the other modules on the Cyborg, i.e., decide what modules should be active at what time. The controller is implemented in Python as a ROS node and uses the ROS actionlib protocol for controlling other modules. Internally it has a state machine to ensure that the NTNU Cyborg is in a well defined state. Furthermore the controller has a model for the Cyborg's moods, e.g., "angry". It uses a PAD emotion state model. The mood can be influenced and used by other modules. If the Cyborg becomes idle, the controller selects activities that will increase the PAD values.

A navigation module for the NTNU Cyborg was also implemented. The module connects to the controller and uses the ROSARNL node for controlling the Cyborg's robot base. This module was selected to be the first module to be integrated with the new controller because it provides features that are relevant for the short term goal of having the Cyborg move around in the campus hallways of Glassården at NTNU. The module provides behavior such as scheduling the Cyborg to be at a certain location at a given time, allowing a user to ask where some locations are and allowing the Cyborg to start wander/moving around in a known location (using ROSARNL).

The controller module, the navigation module and some other minor modules was tested together to see if they worked and to asses if it is a viable solution. The observation showed that the system worked as expected (based on the specifications). The proof of concept works good enough and should be continued developed and used in the NTNU Cyborg. It provides features that are relevant for the short term goal of the project.

Sammendrag

Arbeidet med denne masteren var utført som en del av NTNU Cyborg prosjektet på NTNU. Flere moduler har allerede blitt utviklet for Cyborgen. Disse modulene gir Cyborgen mye forskjellig funksjonalitet som kan føre til konflikter i Cyborgens oppførsel hvis de kjører samtidig. Modulene detekterer hendelser og handler basert på disse. En kontroller modul er blitt laget for å kunne kontrollere de andre modulene slik at en unngår disse konfliktene. Kontroller modulen er implementert i Python som en ROS node og bruker ROS actionlib protokollen for å kontrollere de andre modulene. Kontroller modulen bruker en tilstandsmaskin for å forsikre seg om at Cyborgen er i en veldefinert tilstand. Kontroller modulen har også en PAD humør model som brukes til å finne et humør, som for example "sint", for Cyborgen. Kontroller modulen tar inn tilbakemelding om endringer i humøret fra andre moduler og oppdaterer humøret. Hvis Cyborgen er inaktiv, velger den aktiviteter som vil øke PAD verdiene.

En navigasjons module var også implementert. Denne modulen kan brukes av kontroller modulen. Navigasjons modulen bruker ROSARNL noden for å kontrollere Cyborgens robot base. Denne modulen ble valgt til å bli den første som skulle kobles sammen med kontroller noden fordi at den har funksjonalitet som er relevant for Cyborg prosjektets kortsiktige mål om å ha en robot som kan vandre i Glassården. Modulen gir funksjonalitet som å kunne bestemme hvor Cyborgen skal være til visse tider, lar brukere spørre hvor kjente plasser er og den gjør at Cyborgen starter å vandre eller flytte seg rundt i område.

Kontroller modulen, navigasjons modulen og noen andre moduler ble testet sammen for å sjekke at systemet fungerte og for å bedømme om de lagde delene var en god løsning. Observasjonene stemmer overens med forventede observasjoner basert på spesifikasjonen. Kontroller og navigasjons modulen har funksjoner som er relevante for det kortsiktige målet til Cyborg prosjektet.

Preface

This Master's thesis has been conducted at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. I would like to thank my supervisor, Associate Professor Sverre Hendseth, for his guidance and advice during this thesis. The thesis is also a part of the development of the NTNU Cyborg. I would therefore also like to thank the NTNU Cyborg team, and especially Martinius Knudsen, for letting me undertake this project.

Contents

1. Introduction

The work with this master thesis is carried out as a part of the NTNU Cyborg project at the Norwegian University of Science and Technology (NTNU). The NTNU Cyborg project aims to enable communication between living nerve tissue and a robot. [38] used the following definition for a cyborg (translated into English):

A cyborg (cybernetic organism) is a cybernetic system where biological neurons can process signals from electrical components, with at least one direct feedback connection between the electronics and the neurons.

The NTNU Cyborg project's goal is to create such an organism by cultivating neurons that later will be connected to a robot where the neurons will control simple processes on the robot. However, currently the project is divided into two separate parts: the biological part and the robotic part. The biological aspect will not be described further in this report. The robotic part aims to create a social and interactive robot that will walk around the campus hallways of Glassgården at NTNU. In this report, the term cyborg and robot will be used interchangeably, even though the robot does not yet have a "biological part" integrated into it.

Several modules have already been developed for the Cyborg. These modules include a variety of functionality which may cause conflict if they run at the same time. This report describes a control module for the Cyborg. The control module is responsible coordinating and synchronizing the activity of other modules on the Cyborg, selecting what modules should be active and coordinating the social robot's emotion (e.g. "happy" or "sad") between modules.

1.1 Previous Student Work on the NTNU Cyborg Project

There has previously been conducted several projects (master thesis and specialization projects) on the Cyborg. Martinius Knudsen, a research assistant employed at the Department of Engineering Cybernetics, is the main organizer of student work. The list below will briefly give a status of the Cyborg and some of the previous projects, but note that some parts may have changed because the Cyborg is currently undergoing hardware and software upgrades [41] [31]:

- The Cyborg has a moving robot base called Pioneer LX from Adept MobileRobots. The reason for selecting this robot base is described in [38] and [33]. Some more details about Pioneer LX is given in chapter 2.7. On top of this base is a frame [28] that allows more hardware to be connected. An image of the Cyborg is given in figure 1.1.
- The Cyborg is using ROS (see chapter 2.1), and the argumentation for this is given in [33]. A module that consist of one or more ROS nodes (see chapter 2.1) can give the Cyborg a specific feature.
- Selfie module [17] [40]: The Selfie module allows the Cyborg to take a selfie together with someone (using a camera) and then uploads the selfie to Facebook.
- Follower module [16]: The Follower module allows the Cyborg to follow a human using a Kinect camera. The Kinect camera is currently being replaced by a new camera [31] [32]. This means that this module is no longer working.
- Iris module [30][29]: The Cyborg has an iris, which can be opened and closed, sort of like a box. This module is currently not using ROS.
- The Cyborg has two arms connected to it ([40]). The arms is currently not in use.
- Simon says module [18]: The Simon says module lets the Cyborg play Simon says.
- Trollface module [39]: The Cyborg has a Trollface module. This module displays a face of a troll, can express emotions and gives the Cyborg text to speech capabilities. Due to changes in hardware and a new version of the underlaying technology, this version is no longer being used, but a new version is being developed.
- Communication module [35]: The Cyborg has a Communication module, it has among other things speech to text and the ability to answer questions and tell jokes. Due to upgrades in software and hardware, this module is currently without a microphone.
- The Cyborg is running Xubuntu 16.04, which is Ubuntu with the Xfce desktop environment [41].

The short term goal for the NTNU Cyborg is to have a robot prototype up and running by the summer of 2017. The robot should be able to navigate a known space (i.e. Glassgården at NTNU) and have simple communication capabilities. Further information about the NTNU Cyborg project can be found at [27].

Figure 1.1: A photo of the NTNU Cyborg.

1.2 Motivation and Goal

As previously explained, several modules have already been developed for the robot and these modules include a variety of functionality which may cause conflict if they run at the same time. If two modules sends commands to the same hardware at the same time it might cause inconsistent behavior for the Cyborg. A solution is needed to ensure that this does not happen. This report describes the development of a control module for the Cyborg. The control module is responsible coordinating and synchronizing the activity of other modules on the Cyborg, that is, selecting what modules should be active. It must also coordinating the social robot's emotion (e.g. "happy" or

"sad") between modules.

The goal is to have a controller module, as described above, implemented as a ROS node. This report also implement a navigation module, that uses the controller's interface, to illustrate how modules should connect to the controller. It also place an important part of the proof of concept. The navigation module was selected to be the first module to be integrated into the controller, since it is related to the short term goal of the robot part of the NTNU Cyborg project.

1.3 Distribution of Work

There are several students working on the NTNU Cyborg robot team. The scanning process of the campus hallways of Glassgården at NTNU was conducted in collaboration with Jørgen Waløen [41].

The controller described in this paper consist of several parts. Some of the part's top level design was done in cooperation with Kaja Kvello[36]. A clarification of how the work was distributed follows in the next two paragraphs.

Both students cooperated on deciding to use actionlib for the communication for controlling modules and for using ROS messages for sending of events. The actual code implementation of the controller was done by me alone.

The controller communicates with gatekeepers (separate processes from the controller). Both students cooperated on the design of the "gatekeepers" and how it should be controlled by the controller. The actual code implementation of the hardware "gatekeeper" was done by Kvello alone. Hence, only the top level design and communication protocols of the gatekeeper will be described in this report. The gate keeper code and the related implementation process is discussed in Kvello's report [36].

1.4 Work Not Included in This Report

During the work with this thesis, some of the results obtained were considered not too relevant for this report and were therefore let out.

When starting this report, the Cyborg was running Ubuntu 12.04 and ROS Hydra. Since then, Jørgen Waløen [41] has updated the Cyborg to run Xubuntu 16.04 and ROS Kinetic. Shortly after the upgrade, the SMACH Viewer did not support Xubuntu 16.04 and ROS Kinetic. 3 days where therefore spent on creating a simple replacement that could create a graph image of the state

machine (called State Machine Monitorer), however, the SMACH Viewer is now available and the State Machine Monitorer is no longer needed, but the source code is delivered as a unused module in the controller node.

Since the Trollface module is currently undergoing upgrades, the TTS (Text To Speech) is not available. It was desirable to have some speech output when testing the system, so a new simple TTS ROS output node was created (2 days). This node is used in the system testing and therefor delivered, but is only intended as a temporary solution.

A simple command line tool was made for testing purposes (1 day). The tool allows for testing the controller by sending events to the state machine, sending text input and manipulation of the emotion system (setting emotions and turning it on and off). This tool is strictly meant for testing/debugging purposes, it is delivered incase future developers may have use for it.

This report does not cover work related to preparing the Cyborg for stand and standing on the Cyborg stand for promoting the NTNU Cyborg project (2 days).

This report does not cover a commit made to the ROSARNL open source Github repository (for fixing a compilation errors occurring when trying to compile the source code of ROSARNL).

1.5 For the Reader

The implementation is (mainly) written in Python. It is assumed the reader is familiar with basic programming and Python syntax. Some parts of the implementation explains how other modules should integrate with the controller. The integration and examples are shown in Python code.

2. Background

This background chapter will give a description and/or introduction to some of the equipment, tools and theories used in this report.

2.1 ROS - Robot Operating System

The NTNU Cyborg is using the Robot Operating System (ROS). This chapter will give an introduction to ROS to make the reader able to understand the design and implementation part of this report.

ROS is a flexible, open source, framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior. ROS is not a traditional operating system like Windows or Linux. Instead it is what is called a host operating system and provides a communication layer on top of the operating system. The officially supported operating systems, depends on the version of ROS and the version of the operating system, and can be fund at [15] . One of the officially supported operating system is Ubuntu. Documentation for ROS can be found at [23] and instruction on how to install ROS can be found at [25].

2.1.1 The Concepts of ROS

ROS consists of a network of nodes. A node is a process that preforms some processing. The nodes are connected in a peer-to-peer network. Nodes communicate with each other by publishing messages on topics. The network is configured at runtime and controlled by a special node called *roscore*. The roscore node must run before any other node can run. A roscore node can be launched by entering *roscore* in the terminal. One of the ideas of having a network of nodes is to make everything as modular as possible and this way encourage corporation between developers of different robots. ROS organize software (nodes) into packages. The idea is to make software easy to use and distribute.

ROS has several useful command line tools. A command line tool called rqt_graph can display the node network and the communication between the nodes. Some important tools are shown in table 2.1.1. More ROS command line tools can be found at [14].

Command	Comment
rostopic	Displays information about a topic
rosrun	Starts a node.
rosnode	Displays information about a ROS node.
rosbag	ROS bag files, including playing, recording, and validating.
rqt_graph	Displays information about the ROS node network.

Table 2.1: List of useful ROS command line tools.

Nodes can be written in any language, but the easiest is probably C++ or Python, since there exist ROS libraries and several tutorials and documentation for these languages. Python will be used in this report. To create a ROS node, the initialisation function must be called. The source code in Python for a minimum ROS node is:

```
#!/usr/bin/env python
import rospy

# Initialize the node with a name and connect to roscore (master)
rospy.init_node(name="cyborg_node", anonymous=True)

# Keeps python from shuting down.
rospy.spinn()
```

Messages

A message is a simple data structure. Messages support standard types like integer, float, string and boolean. ROS comes with some standard messages that are defined in *std_msgs*, however, it is possible to create new types of messages. A message file has the extension *.msg*, e.g. message.msg. An example of a position message is:

```
float64 x
float64 y
float64 z
```

Messages are defined in the msg folder (see chapter 2.1.2). All nodes that are using a message that is defined in another node/package, will have a dependency for that node/package (or more precise, for the message inside that node/package), however, the message is independent of programming language and ”the build system” takes care of creating the message types for the used language (i.e., the nodes can use different programming languages).

Topics

Topics are channels where nodes can publish messages. A topic can have multiple publishers. Other nodes can then subscribe to the topic. When a message is published, all the nodes that subscribes to that topic will receive the message. This message system allows the nodes to easily communicate with each other. The Python code, for basic usage of publishers and subscribers is:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
rospy.init_node('talker', anonymous=True)

# The Publisher Part - Create a publisher and send a message
# If a buffer overflow occurse, the oldest unsendt message will be removed.
pub = rospy.Publisher("node_name/topic_name", String, queue_size=10)
pub.publish("Hello, World!")


# The Subscriber Part
def subscriber_callback(data):
    rospy.loginfo(data.data) # Executed when a message is received

# Create a subscriber that is lisning to a topic
rospy.Subscriber("node_name/topic_name", String, callback)
# ROS spin() must be called to see if new messages is available. If there is,
   the callback function (set to handle the topic) will execute. Spinn will
   loop and therefore block until ROS terminates.
rospy.spin()
```

The command line tool *rostopic* [9] can inspect running topics and messages at runtime.

Services

Services are the way that ROS handles requests and response communication. A node can act as a server and offer services to other nodes. A simple example of such a service is an addition service where the server would receive two numbers and add them together and replay with the sum. Services uses messages similar to the publish/subscriber model, but the message consist of to parts, one request part which is sent to the server and one response part which is sent back to the calling node. The two messages are created in a single *.srv* file, but separated with a —. Self

created service files are located in the a *srv* folder (see chapter 2.1.2). An example of a service file is:

```
float64 a
float64 b
---
float64 sum
```

Service calls are blocking calls, i.e. when a client node is calling a service from a server node it will send the request message and block until it receives the replay message. The Python code for using services are shown below. Documentation can be found in in [10].

```
# Service server
def callback_add(req){
  res.sum = req.a + req.b;
  return res;
}
service = rospy.Service("/node_name/add", Add, callback_add)
rospy.spin()

# Service client
rospy.wait_for_service("/node_name/add")
try:
    add = rospy.ServiceProxy("/node_name/add", Add)
    sum = add(a=1, b=2)
    print(sum)
except rospy.ServiceException, e:
    rospy.logdebug(str(e))
```

The command line tool *rosservice* can list services and call them at runtime [8].

Actions

Service calls are, as mentioned in chapter 2.1.1, blocking until a response is received. This can be undesirable if the requested service takes a long time. Actions are similar to services, but they can be none-blocking and provide feedback or status on the progress of the action. Actions can also be canceled or changed while it is in progress. An example of a use case is when a node controlling a robot is calling a node to move an arm to a given position, the controller node can call the arm-node and then do other operations while it waits for the arm to reach its destination. The arm-node can, but don't have to, provide feedback about its current position to the controller node. When the

arm-node reaches its final position, it will send the response back to the controller node. Since the sending and receiving of these messages is non-blocking, callback functions are used.

Actions are located in the *actionlib* package. Actionlib uses a client-server architecture and the ActionClient and ActionServer communicate using a ROS *Action Protocol*. This protocol is built on top of ROS messages. An action message file (*.action*) consist of three parts, an order, a result and a feedback part, defined in a action message file. The parts are separated with —, similar to service files. Action messages are located in the action folder (see chapter 2.1.2). An example is:

```
# Order: To new posisition
int64 new_x
int64 new_y
int64 new_z
---
# Goal: This is where the robot ended
int64 final_x
int64 final_y
int64 final_z
---
# Feedback: The arm is moving, and we are currently at
int64 current_x
int64 current_y
int64 current_z
```

Actionlib provides an API for the user to set up the server and client side. After the server has received a goal, it may be in one of several states. The client side will also be in a corresponding state. A server-state-transition diagram is shown in figure 2.1.1 and a client-state-transition diagram is show in figure 2.1.1. There are many states-transitions shown in the images. Some important part to note, is that when the client sends a goal, the server will receive the goal. The goal may succeed, be aborted or be canceled by the client. This means the client's goal, may succeed, be aborted or the client may choose to preempt(cancel) its own goal, either way, it is important to handle all cases. On server side, this means that it is important to check if the goal is being canceled by the client. An example is shown below and more information about the *actionlib* can be found at [1]. The example is not a full executable example, but shows some of the key aspects of actionlib.

```
import actionlib
from arm_node.msg import RobotArmAction

```

```
# The action server side
# Called when the client connects to the server
def arm_callback(self, goal):
    # Move the arm to position goal.x, goal.y, and goal.z
    while not rospy.is_shutdown():
        if completed: # If robot arm arived at goal
            server.set_succeeded(server_result)
            return
        if error: # If unable to move arm to possition
            server.set_aborted()
            return
        if self.server.is_preempt_requested():
            server.set_preempted()
        else:
            # Continue moving...

server = actionlib.SimpleActionServer("node_name/action_name", RobotArmAction,
    execute_cb=arm_callback, auto_start = False)
server.start()
rospy.spin()

# The action client side
client = actionlib.SimpleActionClient("node_name/action_name", RobotArmAction)
if (client.wait_for_server(rospy.Duration.from_sec(5.0)) == False ):
    rospy.logwarn("ERROR: Unable to connect to server.")
    # ABORT HERE!
else:
    goal = RobotArmGoal()
    goal.x = 34.70 # ect ...
    client.send_goal(goal, client_done_callback, client_active_callback,
    client_feedback_callback)
    # Do robot stuff. The callback functions (above) (not defined here) are
    called when the goal completes, becomes active and when the server sends
    feedback.
    # If the client needs to communicate with the server there are several
    option, some of them are:
        # client.cancel_all_goals()
        # client.set_prempted()
```

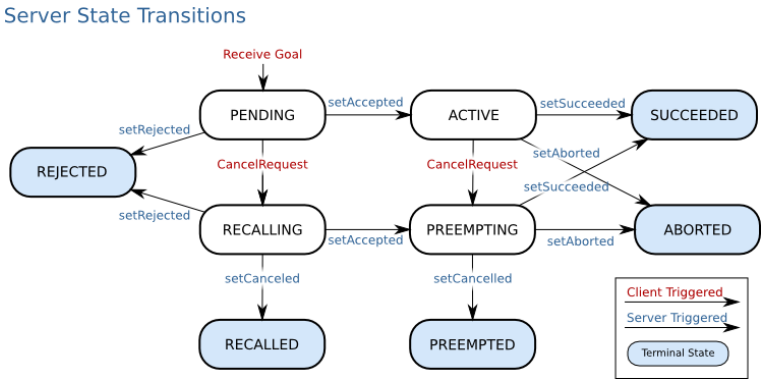

Figure 2.1: Shows the server state transition. The image is taken from [24].

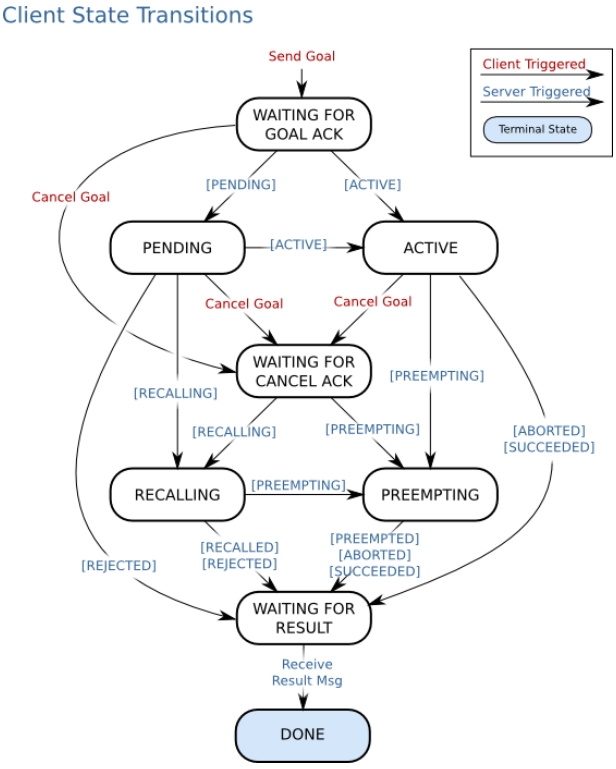

Figure 2.2: Shows the client state transition. The image is taken from [24].

Parameter Server

A parameter server is shared between nodes and is accessible via the ROS network. The server can be used for storing and retrieving parameters (that are globally viewable) at runtime. A common use case is system configuration parameters and state information. Documentation about the parameter server can be fund at [6]. The Python code for getting and setting parameters at the server is:

```
rospy.set_param("/global_parameter_name", "Cyborg")
name = rospy.get_param("/global_parameter_name")
print(name) # Cyborg
```

The command line tool *rosparam*, enables getting and setting of parameter values at runtime.

2.1.2 Catkin - The Build System

ROS is using Catkin as a build system. Catkin requires a catkin workspace that has a *src* folder containing ROS packages to be built. Each package must contain a CMakeList.txt file (see chapter 2.1.2), a package.xml (see chapter 2.1.2) file and a *src* folder. A package may also contain *msg*, *srv*, and/or *action* folders, if it contains self created msg/srv/action messages. A workspace can be created with the command line tool *catkin_init_workspace* and a package can be created with *catkin_create_pkg ¡package_name¿ [depend1] [depend2] [depend3]*. To compile the packages in the workspace, the command line tool *catkin_make* can be used. It must be called in the workspace folder. Workspaces are described in more details in [5] and catkin is described in [2].

CMakeList.txt

Cmakelist is documented at [3]. Cmakelist contains information about how catkin should compile a package. If a package is generated with catkin it will contains a default cmakelist file, however, everything inside the file is commented out, so the file must be edited. A catkin file must contain the required CMake version, the package name, build dependencies, what messages/services/actions to generate, package build information and included libraries.

Package.xml

The package.xml file contains information about the package. It minimum contains the package name, version number, a description, the maintainer, the license and all runtime and build depen-

dencies. The package.xml file is documented at [4].

2.1.3 Running a Node

Before any other node can run, the roscore node, must run. Roscore can be started by typing roscore in the terminal. When roscore is running a new node can start running by using the *rosrun pacakge_name executable_file* command line tool. ROS has a convenient way to launch multiple nodes, called roslaunch. The roslaunch command line tool take in a package name and a roslaunch file. The file can contain default parameter to read at startup, and it can contain other nodes that should be launched. Roslaunch will automatically run roscore if it is not already running. Roslaunch files are documented at [7]. A simple example of a ROS launuch file that launches two nodes is:

```
<launch>
<node name="cyborg_node1" pkg="cyborg_node1" type="node1.py" output="screen"/>
<node name="cyborg_node2" pkg="cyborg_node2" type="node2.py" output="screen"/>
</launch>
```

2.1.4 ROS and the NTNU Cyborg

The NTNU Cyborg is using ROS. The reason for selecting ROS is described in [33]. Some important aspects are that a network was a desired solution for keeping things modular and ROS already had such a solution. ROS also has a lot of documentation, is open sourced and has a large community.

2.2 State Machines

A finite-state machine (FSM) is a method of modelling a system composed of a finite number of modes. The way the system behaves, depends on the mode it is in, and the state machine can at any given time be in exactly one mode. In a state machine, these modes are called states. The state machine can change the state based on external events (inputs). The next state is determined by the current state and the event. The change from one state to another is called a transition. There are several types of state machines, e.g., Mealy and Moore state machines are example of basic state machine, while a Harel and UML state charts are examples of more complex state machines. Some example of places where finite state machines are used are in modelling of application behavior,

design of hardware digital systems, software engineering, compilers, network protocols and vending machines. Figure 2.2 shows an example of a very basic state machine consisting of two states, *State 1* and *State 2* and the possible transitions between them. The transitions are caused by *Event 1* and *Event 2*.

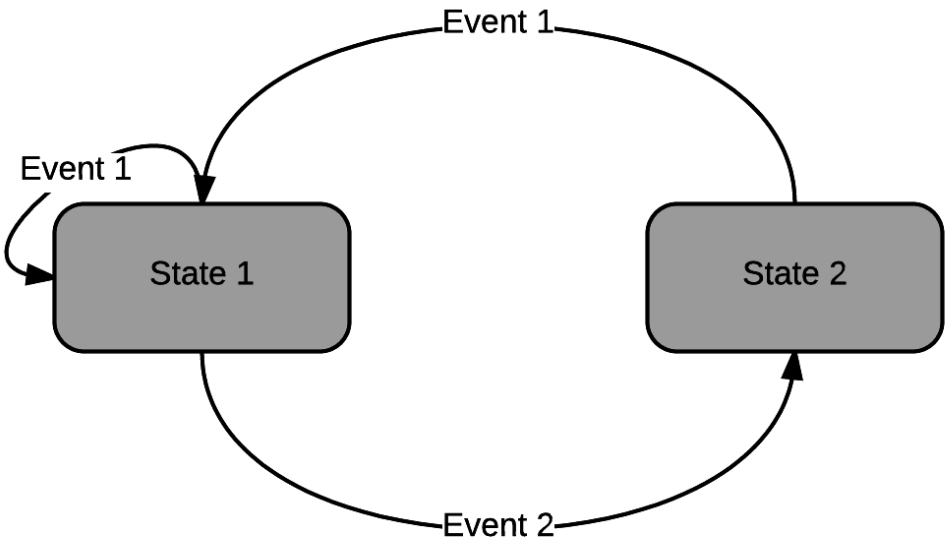

Figure 2.3: *Shows an example of a state machine.*

2.2.1 The Moore State Machine

The Moore machine is named after Edward F. Moore, who presented the concept in a paper in 1956. The Moore state machine is one of the basic state machines. In a Moore state machine the output depends only on the present state.

2.2.2 The Mealy State Machine

The Mealy machine is named after George H. Mealy, who presented the concept in a paper in 1955. The output of a Mealy machine is determined both by its current state and the current inputs. The Mealy state machine is one of the basic state machines.

2.2.3 The Harel State Chart

Harel state charts is a finite state machine that supports hierarchy, parallelism and broadcasting. In Harel state charts, a state can consist of a hierarchy of sub states. Parallelism means that two or more state machines can be drawn in the same diagram and run simultaneously. In addition, the state machines can communicate with each other through broadcasting. One of the state machines can broadcast an event for the other state machine. The Harel state charts has been adopted by the UML state machine standard.

2.2.4 Implementation Paradigmes

There are many ways of implementing a state machine in software. One approach is to use a if-else approach (switch) or a table to store all transitions between states in one central place. A transition table shows what the next state based on the current state and the event. An other approach is the state pattern where the state machine is implemented in an object oriented way. In this pattern, each state is derived from a state superclass (which has a specific interface that the state machine class is using to evoke the state) and each state is responsible for determining the next state.

2.3 SMACH - A State Machine Library Written in Python

SMACH stand for *State MACHine* and is a task-level architecture for rapidly creating complex robot behavior. It is written by Jonathan Bohren and is licensed under BSD, [11]. SMACH can be used for creating finite state machines (FSM). It supports concurrency and hierarchies of state machines. It is written in Python and the core is independent of ROS, however it has many features that integrates with ROS and there are ROS tools that can be used with SMACH. This chapter will give an introduction to SMACH, to make the reader able to understand the implementation part of this report.

2.3.1 How to Use SMACH and Where to Find More Documentation

More information about SMACH can be found in [11]. To create a state machine with states in SMACH, one first have to create some states. A State is a class and can be created from the $smach.State$ class. A new state class that inherit from the $smach.State$ class, must implement an $execute(self, userdata)$ method. The $execute$ method is called when the state ma-

chine enters the state and the method must block until the state is finished and a new state is ready to execute. The execute method returns an event/trigger/outcome of the state. The state machine will use the event to determined the next state. The event that is returned, must lead to a state. If the event is not defined, the behavior is undefined. A state machine can be created with $smach.StateMachine(outcomes)$ method. A state machine can get states and other state machines added to it. When adding a state machine to a state machine, it will work like a hierarchy of state machines with sub states. A state machine is added the same way as a ordinary state with *smach.StateMachine.add(label="STATE_NAME", state=StateClass, transitions=transitions, remapping=None)*. The transition argument can be on the format {*"event1":"STATE_NAME_1", "event2":"STATE_NAME_2"* ...}. A small example is given below:

```
class Foo(smach.State):
    def __init__(self, outcomes=['event1']):

    def execute(self, userdata):
        return 'event1'

state_machine = smach.StateMachine()
with state_machine:
    smach.StateMachine.add('STATE1', Foo(), transitions={'event1':'STATE1'})
state_machine.execute() # Execute the state machine
```

2.3.2 SMACH and the Userdata Concept

Sometimes it is necessary to transfer data between states, e.g. if one state produces a mathematical result that is to be used in the next state. In SMACH this can be done with the $userdata$ concept. The execute method takes in a $userdata$ parameter, provided by the state machine. The states can have input and output keys. These can be mapped to the state machines input and output keys. This way, the $userdata$ can be transferred between states. To make these connections, the $smach.StateMachine()$ method can take in $input\_keys$ and $output\_keys$ arguments. The arguments are simply lists of strings (key names). The $smach.StateMachine.add()$ method can then take in a $remapping$ argument. This argument is a list of corresponding keys. It uses the format: *remapping={"state_input":"state_machine_data", "state_output":"state_machine_output"}*. When the mapping is done, the $userdata$ can be accessed in the $execute$ method with $userdata.key\_name$. More information about the $userdata$ concept can be fund at [12].

2.3.3 SMACH Viewer - A ROS Package for Viewing a SMACH State Machine at Run Time

A state machine created with the SMACH frame-work, can be inspected graphically at runtime with a tool called SMACH Viewer (SMACH Viewer require ROS). SMACH Viewer can show all states in a state machine (included its hierarchy), all valid transitions between states and its current state(s). The tool can be launched from the command line with $rosrunsmach\_viewersmach\_viewer.py$ For SMACH Viewer to be able to see the state machine, the introspection server must be started. The code listed below shows how this can be done. Further information about SMACH Viewer can be found at [13].

```
# Create and start the introspection server
sis = smach_ros.IntrospectionServer('server_name', state_machine, '/SM_ROOT')
sis.start()
state_machine.execute() # Execute the state machine
rospy.spin()
sis.stop() # Stop the introspection server
```

2.4 Database and SQLite

A database is a collection of information that is organised so that it can easily be accessed, managed, and updated [21]. Python comes with SQLite ($importSQLite$). SQLite is a self-contained, high-reliability, embedded, full-featured, public-domain, SQL database engine. More information about SQLite can be found at [22].

2.5 The PAD Emotional State Model

The PAD emotional state model is a psychological model developed by Albert Mehrabian and James A. Russell to describe and measure emotional states [19]. It has been applied to consumer marketing and virtual agents. According to [26], the PAD model uses a three dimensional emotion space to measure emotional states. The dimensions are pleasure/displeasure (P), arousal/non-arousal (A) and dominance/submissiveness (D). Pleasure/displeasure distinguishes the positive/negative affective quality of emotional states, arousal/none-arousal refers to a combination of physical activity and mental alertness, and dominance/submissiveness is defined in terms of con-

#	Name	Pleasure (value)	Arousal (value)	Dominance (value)	Threshold
1	Angry	-0.51	0.59	0.25	0.1
2	Bored	-0.65	-0.62	-0.33	0.1
3	Curious	0.22	0.62	-0.10	0.1
4	Dignified	0.55	0.22	0.61	0.1
5	Elated (Happy)	0.50	0.42	0.23	0.1
6	Inhibited (Sadness)	-0.54	-0.04	-0.33	0.1
7	Puzzled (Surprised)	-0.41	0.48	-0.33	0.1
8	Loved	0.89	0.54	-0.18	0.1
9	Unconcerned	-0.13	-0.41	0.08	0.1
10	Neutral	-	-	-	-

Table 2.2: *Emotional States*

trol versus lack of control. In this model, emotions (or feelings) can be described by these three variables. An example is the emotion of anger, which can be described a state of experience strong displeasure (-P), strong stimuli (A+) and a sense of enough control (D). On the other hand an emotion like board is described as a state of experience strong displeasure (-P), little stimuli (-A) and lacking control (-D). The PAD values ranges from -1 to 1. In the PAD model, each emotion correspond to a specific value. An example is the emotional state of angry, which corresponds to the value (-.51, .59, .25). Several emotions can be found in [26] and are listed in table 2.2.

2.6 Motivating a Robot by Using Rewards

[34] presents a robot motivational system design framework where an agent uses drives and the rewards from preforming actions to select what action to preform. A drive, d_i, is a motivational unit that describes a robots purpose and the reason for action selection. A system can have many drives and each drive d_i has a satiation level $\sigma_i \in [0, 1]$. 0 means that the drive is starved and 1 means the drive is fulfilled (and should not effect the selection of action). Each drive has an associated priority function $p_i(\sigma_i)$. The priority function tells something about how high priority the reward for each drive has and is based on the current value of σ_i. For example if σ_i is low, p_i should be high. p_i can also tell something about the future availability of d_i. In a known environment, where an agent is in a known state and each time step t preforms an action a, that

results in a known change in σ_i and leads to a new known state, then the reward for each drive given that a is selected is given by equation 2.1, while the total reward for a is given by 2.2. The idea is that the action a with the highest reward R_a gets selected.

$$r_i(t+1) = p(\sigma_i) \cdot (\sigma_i(t+1) - \sigma(t)) \tag{2.1}$$

$$R_a = \sum r_i \tag{2.2}$$

2.7 The Cyborgs Robot Base: PionerXL from Adept MobileRobots

The Pioneer LX is a general-purpose, indoor mobile robot platform. It's made by Adept MobileRobots. The Pioneer LX is the robot base of the Cyborg, and is used for moving the Cyborg around. It has a built in computer that runs Xubuntu 16.04. The base is equipped with laser rangefinder sensor, ultrasonic (sonar) sensors, and a bumper panel. Software provides the capability to know where the robot is located within an indoor workspace, and to navigate safely and autonomously to any accessible destination within that workspace, continuously and without human intervention. The Pioneer LX comes with a range of software as well as a full C++ SDK:

- ARIA is a core development library for controlling the robot.
- arnlServer is a program that can be used with for example MobileSim or MobileEyes.
- ARIA Demo is a software demo that allows a user to control the robot with a joystick or through keyboards.
- sickLogger is an ARIA program. It can connect to the robots joystick and can create a $.2d$ scan of an area, which later can be used in Mapper3 to create a map.
- Mapper3 is a tool for processing and editing maps that the robot can use. It uses the 2d scanned files to create a $.map$ file. The robot base can then use the map. When editing maps, it's possible to create restricted areas, fix missing walls and remove "notice" from the scanned area (e.g. people that was in the area when scanning).
- MobileEyes allows a user to connect, remotely or locally, to the robot base. It can control the base (sending it places, control speed) and see where it is on map (if it uses a map), see battery level, make configurations and more.

- MobileSim is a simulator software for use with ARIA, ARNL, SONARNL, or other software. It can simulate the hardware. The ROS node ROSARNL can for example connect to MobileSim instead of the hardware.
- ROSARNL is a ROS node that other ROS nodes can use to interface the hardware (e.g. instead of using ARIA library).

2.7.1 Scanning and Creating Maps for the PionerXL

The Pioneer XL and the related software can be used to create a map of a location. A map is needed for navigation. The robot base is able to navigate a known location. A map can be created by a 2d scan of an area. The simplest way to scan an area is to use the sickLogger software. The scan is made by starting the sickLogger software and using the joystick to move the robot around in the area that is going to be scanned. The scanning results in a $.2d$ file. *Note that the size of the .2d file depends on the area scanned, not the time used.* The $.2d$ file can then be imported into Mapper3. The Mapper3 creates a $.map$ file from the $.2d$ file. The $.map$ file can be edited if it contains any errors or if there should be any restricted areas where the robot should not move. The map can be ”uploaded” to the base using the MobileEyes software.

2.7.2 ROSARNL - The Cyborg's ROS Interface to PionerXL

ROSARNL is available at Github [20] and more documentation is available there. The ROSARNL package contains a ROS node called *rosarnl_node* which provides a ROS interface to basic ARNL features. It does require the ARNL libraries to be installed and it uses the ROS *base message* type. This message type can be installed with:

```
sudo apt-get install ros-kinetic-move-base
```

Since the ROSARNL software is a ROS package, it can be placed in the catkin workspace's source folder and compiled with $catkin\_make$. When compiling ROSARNL with $catkin\_make$ it seams there is a problem with dependencies on some ROS messages, causing the compilation to fail with errors. This is most likely caused by some messages that has not yet been generated. The solution is to try to compile a few times, since the error messages will disappear when the missing ROS messages get created. The ROSARNL node can be started using rosrun:

```
rosrun rosarnl rosarnl_node
```

The ROSARNL offers many features, some of them is using actionlib to move the base to coordinate on a map and one is a ROS service to tell the base to start wandering. The ROSARNL action server for moving the base is */rosarnl_node/move_base*. It uses a standard *MoveBaseAction* message and the wandering is a service call available at */rosarnl_node/wander* using a standard empty message.

3. Requirements and Consideration for the Controller Module

The problem description in this thesis was very broad and open on how the problems should be solved. Therefor, an analysis of the system that existed and the problem was needed. This chapter explains the problem (in more detail) that the controller module must solve and goes through some considerations, requirements and constraints for a controller for the NTNU Cyborg. These considerations result in the specifications in chapter 4. In this report, the following definitions are used:

- **Module:** Some hardware or software that provides some functionality/feature for the Cyborg through a ROS node.
- **Input Module:** One or more ROS nodes that takes input from hardware and make the data (in some form) available for the ROS network, i.e., for other ROS nodes. An example is a ROS node that takes sound input from the microphone and publishes the text on the ROS network.
- **Output Module:** One or more ROS nodes that ”directly” control the hardware (through drivers) (and often makes it accessible for the other ROS nodes). An example is a node that takes in text and provides a voice output through the speakers.
- **Behavior Module:** One or more ROS nodes that provide some functionality that adds specific behavior to the Cyborg, e.g., the Selfie module. It may for example receive data from an input module, and then tell an output module to do something, but it does not directly interact with the hardware (drivers).
- **Event:** Something, often external, that has happened, been detected by a behavior module and that the module wants to act on, e.g., a user rising an arm (signaling the user wants the Cyborg to following it).

3.1 Description of the Problems that the Controller Module Must Solve

A social and interactive robot needs to have features and functionality that make it interesting. Some papers regarding the NTNU Cyborg and these topics are [37][39][35]. All features that are developed for the NTNU Cyborg are implemented in modules separate from the controller module. The Cyborg is using ROS and all modules may consist of one or more ROS nodes. The modules are detecting events and acting on them. A downside to having many independently running modules, is that modules may have conflicting interests, e.g., one module may want to take a selfie with someone, while an other module may want to walk away. When all modules run independently, this can be a problem. Figure 3.1 shows modules communicating over ROS without any controller.

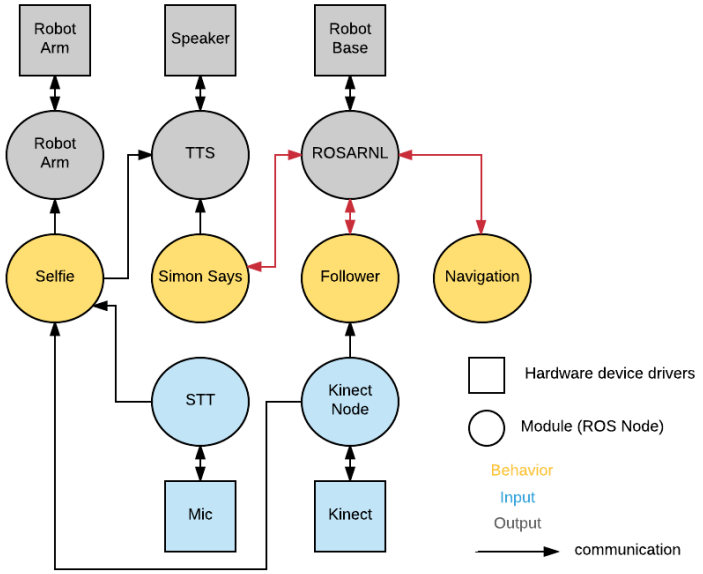

Figure 3.1: *Shows a scenario of modules and the communication between them. The main problem is for the output modules. There may (easily) be situation where an output modules may receive input from multiple behavior modules. The input may be conflicting. As shown in the image, the ROSARNL module (which controls the robot base) may receive "move left" from the Follower module and then receive "move right" from Navigation module, this may cause undesirable behavior in the Cyborg. In figure 3.1, this communications are indicated with red arrows. An other example that can become clear from the figure, is that the Selfie module has not registered with the base, witch means that the base may start wander off when the Selfie module is trying to take a selfie.*

A controller is needed for controlling the Cyborgs behavior, so that it behaves in a consistent manner, i.e. the controller must decide what modules is active, and which are not. The controller

must be able to preempt modules, so that modules with a higher priority may take over instead. It's also important to have a system that ensures that the Cyborg is not remaining idle, e.g., when no modules are asking for control. The Cyborg need a motivational system. The motivational system can work the same way as the modules event system work. Modules can provide the motivator with events that it can select. The motivator must be aware of the Cyborg's current state and what actions that are possible in that state.

But what can motivate the Cyborg? When the Cyborg is in a state such as idle, there can be many actions the Cyborg can preform. There should be a reason for selecting one action over an other. An approach is to use a reward system.

The sort term goal of the robot part of the NTNU Cyborg project, is to have a social robot that is able to navigate a known location. The Cyborg has a module for displaying facial expressions, such as smiling or sadness, however, as described in [39], it does not have an internal emotional state. The controller module must provide an emotional model for the Cyborg and a way for the modules to influence the emotional state, as well as a way of coordinating the social robot's emotion (e.g. "happy" or "sad") between modules.

3.2 Selecting Communication Protocols for Controlling the Behavior Modules

There are many ROS protocols to select from, e.g., publisher/subscriber, services, actions and parameter server. For sending events to the controller a, publisher/subscriber architecture is desirable because it allows the controller to subscribe and receive events from multiple modules. The ROS actionlib protocol (chapter 2.1.1) was selected for controlling what behavior modules are active because:

- It is a ROS protocol.
- It has a server-client architecture that allows the controller (client) to request action from module (servers).
- It is possible to get feedback on ongoing actions.
- It is possible for the controller to preempt any action.
- It is none blocking.
- The other ROS protocols does not provide all these features.

3.3 Constraints for ROS, 3rd Party Modules, Output Modules and Gatekeepers

An important consideration is that some hardware from 3rd parties may come with support for ROS. It may not be desirable or possible to change the internal working of these modules. Input modules does not need to be controlled by the controller as long as the input modules does not directly control the Cyborgs behavior. If a module is an output module, it may not be necessary for the controller to control them, as long as all the behavior modules that control the output module is controlled by the controller. However, there may still be desirable to have some sort of gatekeeper functionality for the output modules. The gatekeeper should receive instructions form the controller and decide where the output module should accept data from. A consideration for the gatekeeper is that it must be possible to add it between a behavior module and a output module, since the output module may be a 3rd party solution. Figure 3.3 show the gatekeeper concept.

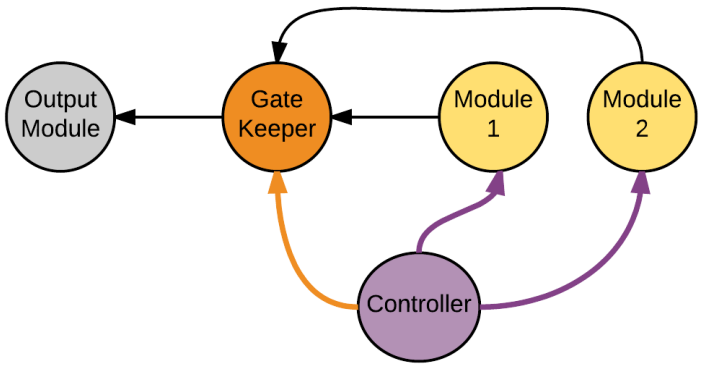

Figure 3.2: *Shows the gatekeeper concept. The gatekeeper stops all messages from modules that are not supposed to communicate with the output module. What module(s) to let trough is decided by the controller.*

3.4 Organizing the Behavior Modules in to a State Machine

If behavior modules have action servers. The controller can use these actions to activate modules. From the controller's perspective, these actions can be organized into a state machine, where each state represent an action. Chapter 2.2 describes several types of state machines. It may at first be enough with a Mealy or More type, however, as the number of modules grows, it may be desirable

to have a Harel state chart (machine). As described in chapter 2.2.3, the Harel state chart supports Mealy, More, parallelism, hierarchy etc. This could be an advantage for future scalability of the Cyborg's behavior complexity.

3.5 Selecting a Model for the Cyborg's Emotions and the Communication Protocols for the Emotion System

For developers who develops new modules for the Cyborg, the emotion system must be optional, in the sense that a module can ignore emotions, although it may affect the Cyborgs behavior (in the sense of lacking emotional responses). The argument for this is that some developers may have other priorities then emotions. The PAD emotion model is described in chapter 2.5. The PAD model is selected because:

- It has many easy recognizable emotions, e.g., happiness, sadness, anger, board etc....
- The emotional state is based on numbers (the PAD values), which could let the modules give feedback inform of changes in the PAD values, and let the controller decide how these changes effect the emotional state.
- A bit simplified, for a robot purposes, the values can be determent by:
  - Is it pleasurable(P)?
  - Is it stimulating (A)?
  - Does the robot have control of the situation (D)?

The controller, must inform all other modules of the Cyborg's current emotion and get feedback from the modules of changes in the emotions. A ROS publisher is selected to broadcast the current emotional state and a ROS subscriber is selected for receiving the feedback.

3.6 Considerations Regarding Complexity of Adding and Removing Behavior Modules

There are many people working on the Cyborg. Some of them, may work on it for a shorter period of time. With this in mind, it would be an advantage if the system makes it easy to add, remove and update the list of modules that are on the Cyborg. Changes made to the system can be done

while the Cyborg is offline. A graphical tool to view what modules are connected and what events will lead to what behavior of the Cyborg, would be nice.

4. Specifications for the Controller

The following specifications was selected based on the considerations in chapter 3. The specifications listed below, does often not specify specific models or protocols (except from ROS) mentioned in the design considerations. This is done deliberately.

1. The controller must be implemented in ROS, i.e. the controller must consist of one or more ROS node(s) and the communication with all modules must use ROS protocols.

    1.1. Modules should not be implemented inside the controller, but run as separate node(s) and offer their functionality to the controller, i.e, from the controllers perspective, all modules (and the behaviour they offer for the Cyborg) is accessed the same way over ROS.

2. The controller must make sure the Cyborg is in a ”consistent state” at all times: Modules that are active can not be in conflict with each other. This can, but do not have to, be implemented as a state machine.

    2.1. All detection of events is done in modules outside the controller, but the module notifies the controller of these events.

    2.2. The controller decides if an event should lead to a state change or not.

    2.3. It must be possible for a module to preempt an other module. The preempt rules is handled by the controller, but set by each state.

    2.4. The controller is responsible for controlling the gatekeepers. See the specifications for the gatekeepers (4).

    2.5. Integration of new modules into the system, as well as removal of modules, must be intuitive and fast, but can be done offline. Such changes should not change the functionality of the coordinator, only the behaviour of the Cyborg. Changes are made by the module developer(s).

3. Optional: Through some sort of a system, preferably graphical, see all possible states and the transitions between the states.

4. Optional: Usage of gatekeepers. A gatekeeper works as a message filtering system. It only lets through messages from specific modules. What modules to allow through is controlled by the controller. The gatekeeper may be implemented inside an output module or between an output module and a behaviour module, as shown in figure 3.3.

5. The controller must have an emotion system. It must keep track of the Cyborg's emotion.

    5.1. The emotion system must provide a way for a module to give emotional feedback to the controller. The feedback should be in the form of changes in emotional values primarily, not directly setting an emotion.

    5.2. Based on the emotional feedback, the controller must set the emotional state. The emotional state should be something like ”sad”, ”happy”, ”angry” etc.

    5.3. The emotion system must inform the modules of the emotional state.

6. The controller must have a motivator. The motivator must motivate the Cyborg to do things when there is no external events, i.e., when there is no modules actively wanting control of the Cyborg.

    6.1. When the motivator selects an action to preform, it must activate the action in the same manner as when modules wants control over the Cyborg, i.e., it must select an event and send it to the controller, where it is handled in the same way as an event from an other module.

    6.2. The motivator must be aware of the Cyborg's current active state and what actions it can preform in that state, i.e., it must be aware of events that it can select which will lead to a state transition (and the performance of a new action).

    6.3. All actions are preformed by the respective modules, not the motivator, i.e., the motivator is only responsible for “motivating” the Cyborg to preform actions, not how the actions are preformed.

    6.4. The motivator should select actions based on a reward system. The motivator will select actions that makes the Cyborg “happier”.

5. Specifications for the Navigation Module

A short term goal for the NTNU Cyborg project is to make the robot able to navigate a known location by the summer of 2017. The controller need some behaviour modules to control, so a navigation module is selected, since it is related to the short term goal. Some specifications for the navigation modules is:

1. The navigation module must use the robot base through ROSARNL.
2. The navigation module may use a map of the location.
3. The navigation module should have a database of some known locations, like the cafeteria and entrance area (and where they are on the map).
4. The navigation module should take advantage of the Cyborgs emotional state and it should provide emotional feedback when it moves around.
5. Some desirable features the navigation module could provide is:
    - 5.1. A user can ask the Cyborg where a location is and if the Cyborg know the location, it shows the user where it is.
    - 5.2. A user can tell the Cyborg to go to a known location.
    - 5.3. Make a schedule for the Cyborg, that is, set places the robot should go to at some given time.
    - 5.4. Let the Cyborg move around based on events from the motivator.

6. Design for the Controller Module

The controller design is based on the requirements and the design consideration described in chapter 3 and the specifications from chapter 4. The controller consist of three parts; A state machine, an emotion system and a motivator. The state machine is responsible for tracking and controlling what goals/activities the Cyborg is preforming and is described in chapter 6.1. The emotion system consist of a model for the Cyborg's emotional state and is responsible for keeps track of the internal emotional state. The emotion system is described in chapter 6.2. The motivator (described in chapter 6.3) is a system for selecting events, so the Cyborg can preform actions without waiting for external events. These self generated events are here after referred to as motivational events.

Figure 6.1 shows a simplified overview of how these three parts is connected, and the communication between them. It also shows that the Controller takes inputs, in form of events and emotional feedback, from modules and that the output from the controller is the system state and emotional state. The controller is a single ROS node, but the internal communication between the three parts is going over ROS protocols. More details about the communications and the three parts is given in their respective chapters and chapter 6.4 gives a more detailed overview of how everything is connected.

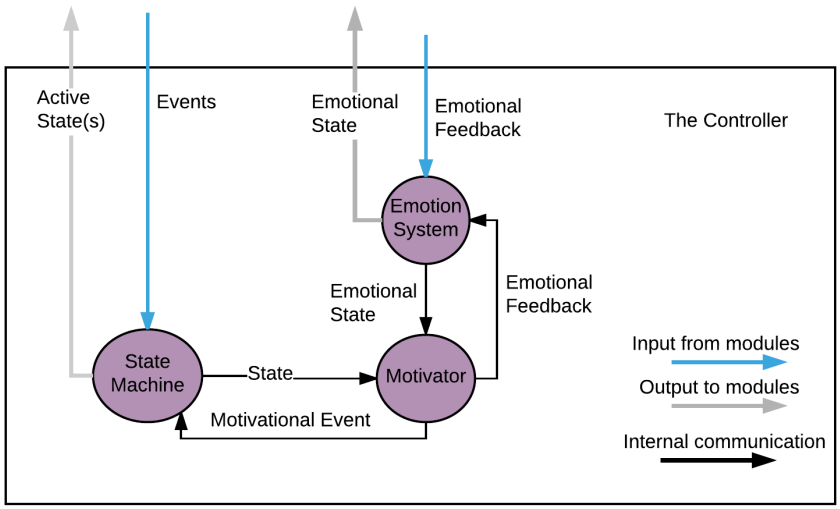

Figure 6.1: *Overview of the internal workings of the controller.*

6.1 The State Machine - Organizing the Behavior Modules Action Servers Into States

A state in the state machine represent an action in ROS. The action is made available by a behavior module using the ROS actionlib protocol. When the state machine enters a state, it contacts the behavior module, using the actionlib protocol, i.e., the state is the action client and the behavior module offer the action server. A behavior module may offer several action servers, where each action server is represented as a single state in the state machine. Figure 6.2 shows the relationship between modules, ROS nodes, action servers, states and the state machine.

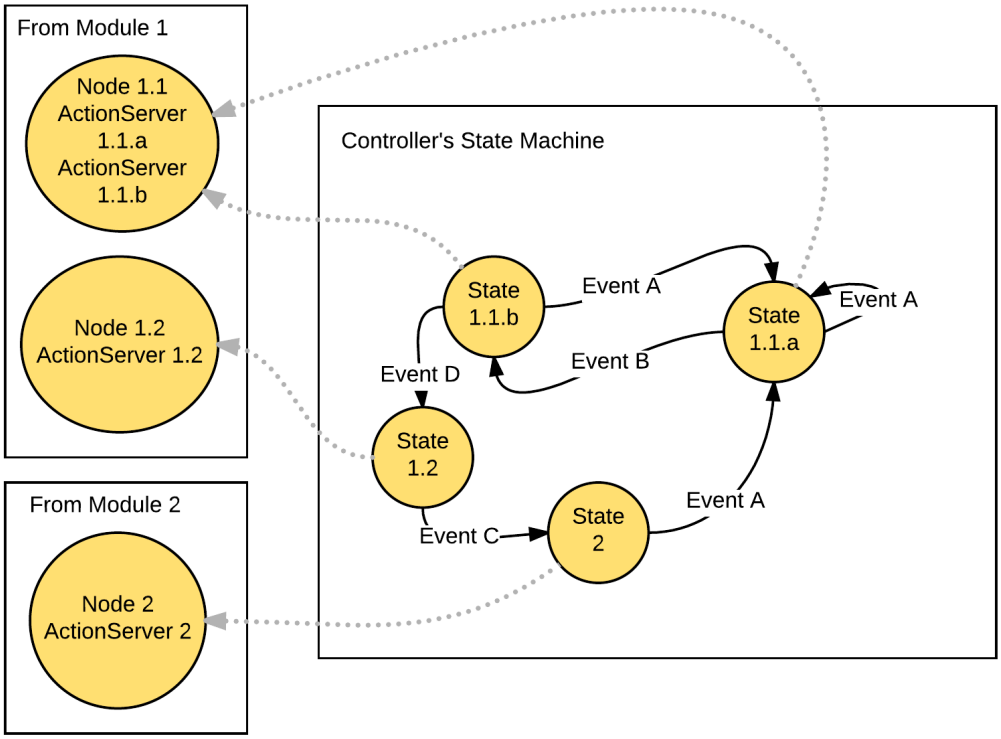

Figure 6.2: *Shows how states in the state machine is related to action servers in modules.*

The state machine is event driven. When a module detects an event, it can send the event name to the controller's state machine using a ROS publisher. The State machine has a ROS subscriber that listens for events. If an event leads to transition between two states, the state machine will change the active state.

As described in chapter 2.1.1, the actionlib server and client can be in one of several states (for

clarification; states in the actionlib protocol, not states in the controllers state machine). Figure 6.3 and 6.3 shows how the communications between a module and the state machine (controller). Figure 6.3 shows the first stage. At first module 2 is active, i.e., the controller's state machine is in a state that is connected to the action server in module 2. Module 1 detects an event and (1) sends it to controller's the state machine. The state machine receives the events, and checks if it leads to a state transition in the state machine (and it does). The state machine then sends a preemption (2) to module 2, which replays with setting the actionlib state to preempted (3). The state machine then changes its internal state by leaving state 2 and entering state 1, as show in figure 6.4. The state machine then connects (4) to module 1 and module 1 replays that the state is active (5).

If the action server sets the actionlib protocol to be in either the succeeded or aborted state, the state machine will treat these as regular events, but start after step (3). This means that the behavior module can use succeed state of the actionlib protocol as an event, e.g., the selfie module can used the succeeded actionlib state to go back to the idle state when selfie is taken. Similar the aborted event can be used. If the state machine is unable to connect to an action server, for example because it does not exist, it will act on an event called aborted. The alternative would be to have a undefined behavior. This means that all states must have a outcome called aborted that leads to a valid state. The simplest thing is to return to the idle state. Similar, if the succeed actionlib protocol is used, the succeeded active state in the state machine must have an event called succeeded that leads to a new state, however, the succeeded event is not required.

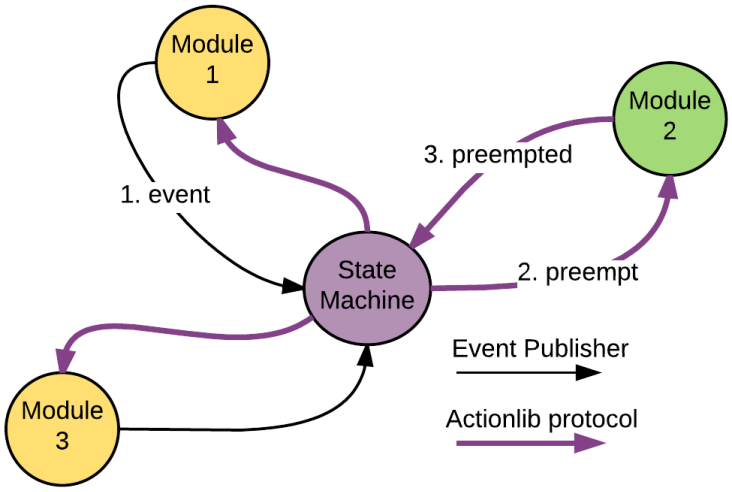

Figure 6.3: *Shows communication between the state machine and the modules.*

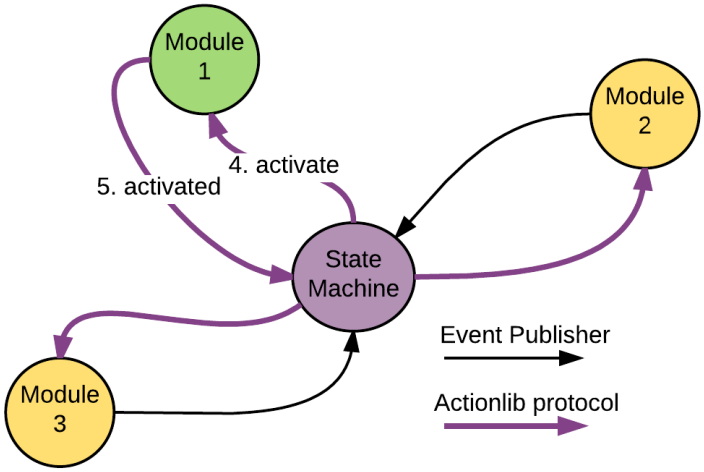

Figure 6.4: *Shows communication between the state machine and the modules.*

6.1.1 Gatekeepers

The system is using gatekeepers to block output modules from receiving instructions from non active modules. Since a behavior module may use several other ROS nodes with different node names other then the behavior module name, each state in the state machine, keeps a list of all ROS nodes that are allowed, while in the respective state, to get through the gatekeepers. This list contains the name of the topic and what resource it has access to. This means that there are several resource lists within the state machine, i.e., one per state. The currently active state's resource list is stored on the ROS parameter server (see chapter 2.1.1). When a state change occurs, the new state updates the ROS parameter server with its own resource list and the state machine publish all state changes over a topic using a ROS publisher (in addition to using the actionlib protocol described above). The gatekeepers subscribes to this topic. When the gatekeepers receive a message about a state change, they connect to the parameter server and get the updated resource list. Figure 6.5 shows the communication and usage of gatekeepers.

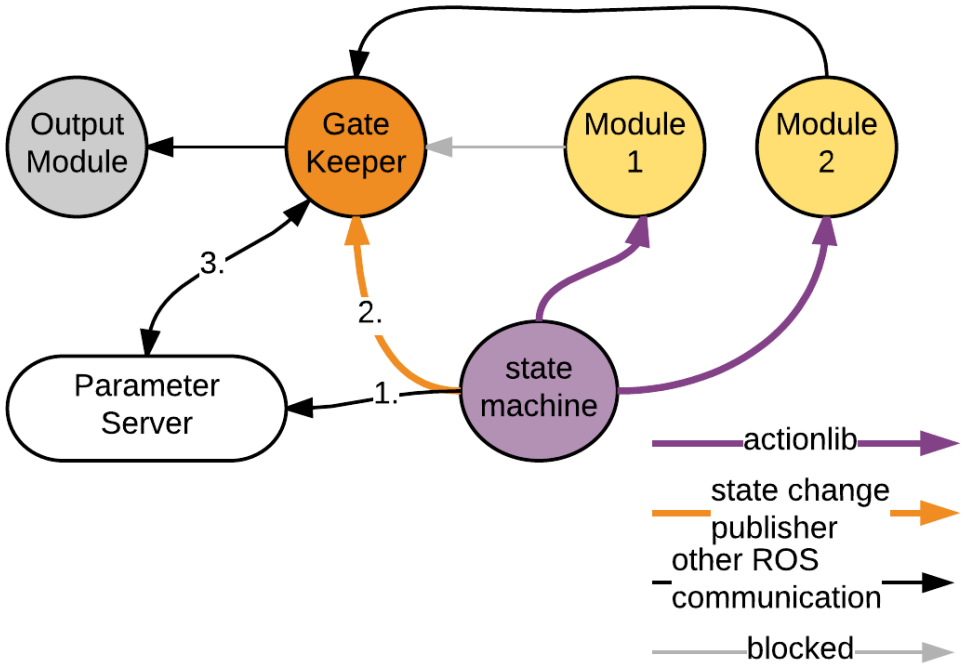

Figure 6.5: *Shows an overview of how the gatekeepers work. The gatekeeper only lets trough messages from nodes specified by the controller (state machine). When a state change occurs, the controller (1) updates the list on the ROS parameter server, (2) the controller publishes the state change and (3) the gatekeeper asks the ROS parameter for the updated list.*

6.2 The Controller's Emotion System

The controller has an emotion system. The emotion system is using the PAD emotion model (see chapter 2.5). The emotion system allows the modules to influence the PAD values. A module can send changes in the PAD values, i.e., ΔPAD, to the emotion system by publishing it on a ROS topic that the emotion system subscribes to. The Cyborg's emotional state will be updated when the emotion system receives changes in the PAD values. Any change in the Cyborg's emotional state is published on the ROS network and, if needed, a module can get the current emotional state through a ROS service. When updating the current PAD values with the feedback received from modules, formula 6.1 is used.

$$E_{next} = E_{current} - E_{decay} + \Delta E_{change} \tag{6.1}$$

$$E = [P_{value}, A_{value}, D_{value}] \tag{6.2}$$

Where E is a three dimensional vector that contains the PAD values, i.e., as shown in equation 6.2. When the emotion system receives a change in PAD values, it would be naturally if the effect from this feedback gradually decays. This is what the decay, E_{decay}, in formula 6.1 represent. The decay is based on the total PAD values, and the decay is towards 0 (as described in chapter 2.5, the PAD values is between -1 and 1). In the controller's emotion system, the formula for finding E_{decay} is given in 6.3.

$$E_{decay} = E_{current}^2 \cdot r \cdot E_{current} \tag{6.3}$$

The equation is listed as it is for clarity. The reason for selecting this formula is that it has the following properties:

- If E_i is the value of one of the dimensions in E, i.e, $i = \{P, A, D\}$, then when $E_i < 0$ the E_i value of E_{decay} is positive and E_{decay} contribution make E_{next} go towards 0 and when $E_i > 0$ the E_i is negative and therefor contribute to make E_{next} go towards 0.
- The formula is based on a decay percentage of the current value. The decay percentage is between 0 and r. The percentage depends on the current PAD values ($E_{current}$). The closer the PAD values, E_i, is to minimum or maximum (-1,1), the larger the percentage gets. This means that the effect of the previously revived ΔE_{change} is decaying slower when the $E_{current}$ is closer to neutral.

- The decay is based on the current PAD values i.e., $E_{current}$.

The emotion system uses a maximum rate of 15%, i.e., $r = 0.15$. The graph for the decay value, E_{decay}, with the selected values is given in figure 6.6.

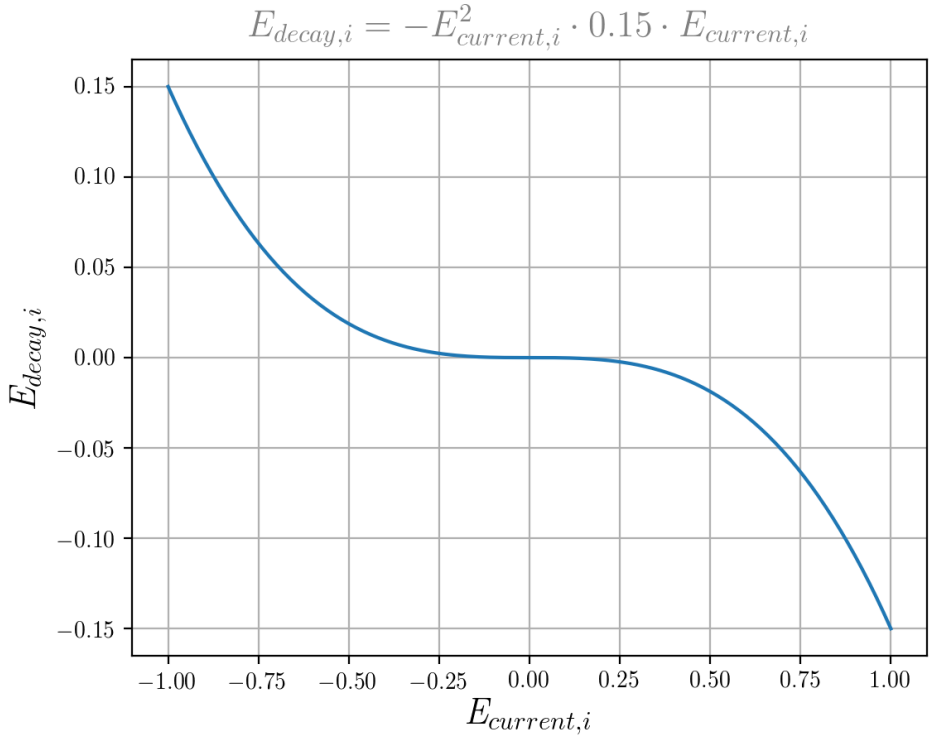

Figure 6.6: *Shows the graph for the decay function.*

6.2.1 The Cyborg's Emotional States

Table 2.2 shows the emotional states the robot can be in and the corresponding PAD values for each state. Since the current PAD value, may not be an exact emotion value, the implementation uses the least squares method to find the closes emotion that is within a threshold, r. If no emotion can be found the neutral state is selected. A three dimensional graph showing the PAD emotional space with the selected emotional state from table 2.2 is shown in figure 6.7.

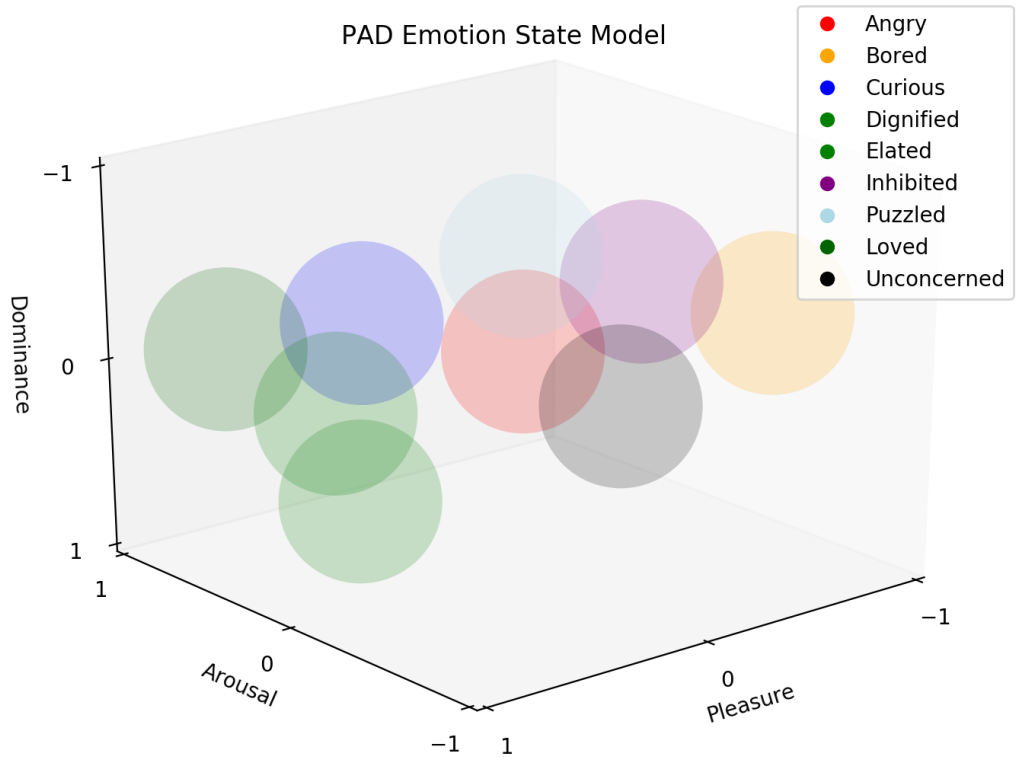

Figure 6.7: *Shows the emotional PAD space and the emotional states.*

6.3 The Controller's Motivator

Behavior modules can detect external events and send them to the state machine and be granted control so it can act on the event. However, some behavior modules may offer activities that the Cyborg can preform without the need for external events, i.e., sometimes the Cyborg can do things on its own without directly reacting to an external event. An example is when it is in the *idle state*, it makes sense that the Cyborg does something instead of doing noting. The motivator is responsible for "motivating" the Cyborg to do activities when there is no external events, such as the user interacting with it. The motivator does this by sending events to the state machine, just like modules. The motivator keeps a list of available events it can select from. The events are added to the list by module developers. What event the motivator is selecting is based on what state it is in and a reward-cost system.

The motivator is using the Cyborg's emotions as a motivating factor. The "goal" for the motivator is to select actions (trough events) that will make the Cyborg happier, i.e., to increase the PAD values. The reward-cost system used is based on the reward system described in chapter 2.6. The three PAD values are selected as three drives and these drives is the base of the reward system. Since the scale used for PAD values [-1,1] and the drives [0,1] is different, the PAD values must be normalized to the drive scale. The normalization is given in equation 6.4. The selected priority function for the PAD drives is $p_i(\sigma_i) = 1 - \sigma_i^3$. The reward r_i for each drive and the total reward R_e for an event, is computer as described in chapter 2.6.

$$\sigma_i = \frac{E_i + 1}{2} \tag{6.4}$$

A problem with using a reward based system is that there is easily possible to think of situations where preforming the same action after each other would yield large rewards. This may be fine in many systems, the "problem" is that the Cyborg is a social interactive robot, which means this is highly undesirable. An example on such a case is that the Cyborg sys "good morning" to the user one hundred times in row to collect the reward. To solve this problem each action (event) has a "social cost" associated with it. This cost works similar to the reward. The event has a saturation, σ_e, between 0 and 1. The saturation level gets filled up when the motivator uses the events and declines over time when the motivator uses other events, i.e., $\sigma_e = 0$ means there is no cost and $\sigma_e = 1$ means there is a large cost. The cost for each event is based on the changes that the event will cause in the PAD drives, $\Delta\sigma_i$ and the events current saturation level, σ_e. The equation used is:

$$c_e(\sigma) = p_e(\sigma_e) \sum \Delta\sigma_{i,e} \tag{6.5}$$

It uses a priority function, just as for the rewards in the chapter 2.6. The priority function for the cost function is selected to be as in equation 6.6. Figure 6.8 shows how the priority functions look like.

$$p(\sigma_e) = \sqrt[3]{\sigma_e} \tag{6.6}$$

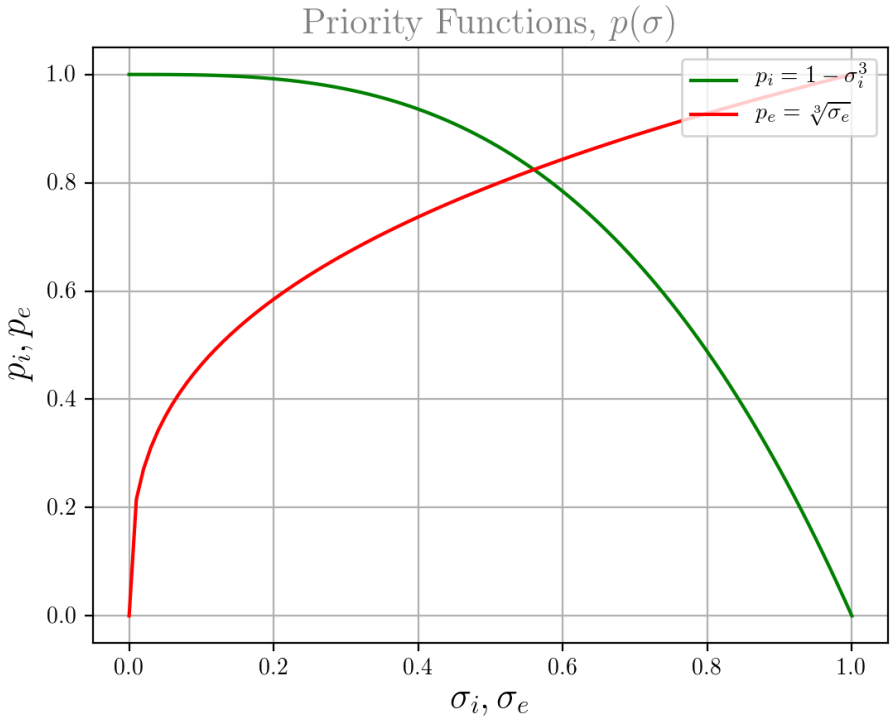

Figure 6.8: *Shows the priority functions for the reward and the cost.*

As described above, the motivator has a list of events it can select from. In addition to containing the associated state, the list contains reward (in PAD) for selecting each event, e.g., when in the idle state there is an event that will cause the robot to move to a new location. This event gives a reward in A. The motivator is subscribing to the topic where the state machine publishes the state changes and it publishes the event on the topic which the state machine listens for events. It also subscribes to the topic where the emotion system publishes changes in the emotional state (and PAD values) and publishes changes in the PAD values when an event is selected.

6.4 Connecting the State Machine, the Emotion System, the Motivator, the Gatekeepers and Modules

Figure 6.9 shows how the state machine, the emotion system, the motivator, the gatekeepers and the modules are connected. The figure (along with the design from the previous chapters) shows that:

- The state machine receives event (over ROS message, blue line in figure) from modules and controls the modules by the actionlib protocol (purple line in figure).
- The emotion system receives feedback in the form of changes in PAD values from modules (over ROS message, pink line in figure). It publishes the current emotion to the modules (over ROS message, yellow line in figure).
- The motivator receives the emotional state from the emotion system (over ROS message, pink line in figure) and the state changes in the state machine (over ROS message, orange line in figure), while it provide feedback to the emotion system (over ROS message, yellow line in figure) and sends events to the state machine (over ROS message, blue line in figure).
- The gatekeeper receives information about state changes from the state machine (over ROS message, orange line in figure) and communicates with the ROS parameter server.

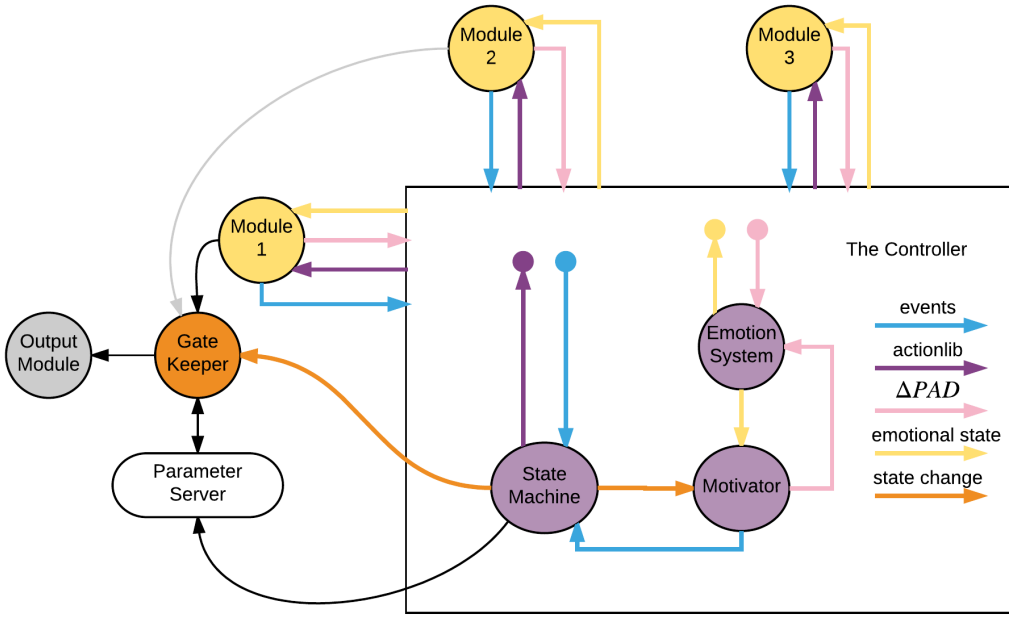

Figure 6.9: *Shows an overview of the system.*

6.5 Behaviour Modules

Behaviour modules are, as described in chapter 3, *modules that provide some functionality that adds specific behaviours to the Cyborg*. A behaviour module must make actions available for the controller. This is done through action servers using the ROS actionlib protocol described in chapter 2.1.1. A behaviour module may have one or more action servers for the controller. If a behaviour module detects an event and wants to be active (i.e. it want one of the action servers to run), it must send the event to the controller using a ROS message. If the controller wants to act on the event, the controller will connect to the action server associated with the event. The controller has a state machine where a state represent an action servers and this state machine determined if the controller connects to the action server or not based on the current state and the event. A behaviour module may subscribe to a topic to receive the emotional state the Cyborg is in. For more information see the controller design.

7. Design for the Navigation Module

The Navigation module is a module that provides navigation behaviour for the Cyborg. The Cyborg has a robot base (described in chapter 2.7). The ROSARNL node provides features that other ROS nodes can use for controlling the base (described in chapter 2.7). The Navigation module uses the ROSARNL node for moving the robot base. The robot base is provided with a map. It also uses the text to speech (TTS) module to give voice feedback to the user, while it receives text input from the communication module (speech to text; STT). This chapter explains how the Navigation module work, it's features and the design. The Navigation module is a single ROS node and all communication with other modules is through ROS. The Navigation module is controlled by the controller, i.e., it offers actions servers that the controller can use. The Navigation module also uses the emotional state provided by the Controller module.

7.1 Requirements

- STT, from for example the Communication module.
- TTS, from for example the Trollface.
- The ROSARNL node.

7.2 Features

The navigation module provide the following behavior for the Cyborg:

- Scheduling of events: It is possible to schedule a date and time when the Cyborg should go to a location, e.g., for going to the cafeteria at lunch time.
- If a user ask what it thinks of a known location, it will replay based on its emotional state.
- If a user ask where a known location is, the Cyborg can show the way.
- A user can command the Cyborg to a known location.
- The Cyborg (controller's motivator) can start moving to known locations or wandering around based on it's emotion.

It is worth pointing out that it uses a very simple and limited keyword search of the text from STT.

7.3 The Database: Locations, Events and Things to Say

The ROSARNL is provided with a map over the area where the Cyborg can move. The ROSARNL node provides functionality for other nodes, such as sending it to coordinates (x,y) on the map. The database in the navigation module contains information about the coordinates for special locations, e.g., the cafeteria and the information desk. The database also contains some ”pre knowledge or assumptions” about these places. A place may be described as ”generally crowded”, i.e., a place where the Cyborg most likely will encounter people, and a place has a descriptive value about how pleasant it is. The pleasant value is used for emotional feedback for the controller. The database also contains calendar events for telling it to go to places at certain times.

To give the Cyborg the ability to talk about some of the locations it has in its database, the database also contains sentences it can use. The sentences does not contain any location names, instead they contains a keyword ”LOCATION” which can be replaced with the actual location name. These sentences are expressing opinion about a location, e.g., ”I hate the LOCATION” or ”I like the LOCATION”. The database stores what type of emotion the sentences is representing. The navigation module can use these sentences to let the Cyborg express an opinion about locations. The navigation module uses the Cyborg's emotional state to determined what it thinks of places, i.e., if it is happy is likes places and if it is angry it dislike places.

7.4 Connecting the Navigation Module to the Controller

The features from chapter 7.2 is provided by three action servers. This means that the navigation module will add three states to the controller's state machine. The states are a planning state, a moving state and a talking state. Several events can lead to these states. The navigation module can detect many events:

- navigation_command: When the modules receives text from the STT, it searches for keywords in the text, if it finds ”go to” or ”move to” and the name of a location that it has in the database, it send a navigation_command event to the controller.
- navigation_information: When the modules receives text from the STT, it searches for keywords in the text, if it finds ”where is” and the name of a location that it has in the database, it send a navigation_information event to the controller.
- navigation_feedback: When the modules receives text from the STT, it searches for keywords in the text, if it finds ”think of” and the name of a location that it has in the database, it send

a navigation_feedback event to the controller.

- navigation_scheduler: If the scheduler finds a calendar event in the database that has started, it sends a navigation_scheduler event to the controller.

In addition to detecting events, the navigation module has one event that is added to the motivator, i.e., navigation_emotional. This way the Cyborg can start moving around without any user or scheduled event. Where it moves depends on the emotional state of the Cyborg. The three action servers may also generate events such as succeeded, aborted, navigation_start_moving and navigation_start_wandering. If the controllers state machine had only a idle state and a conversation state (modules), the navigation module could be connected as show in figure 10.2. The figure also shows that the planning state always comes before the moving state.

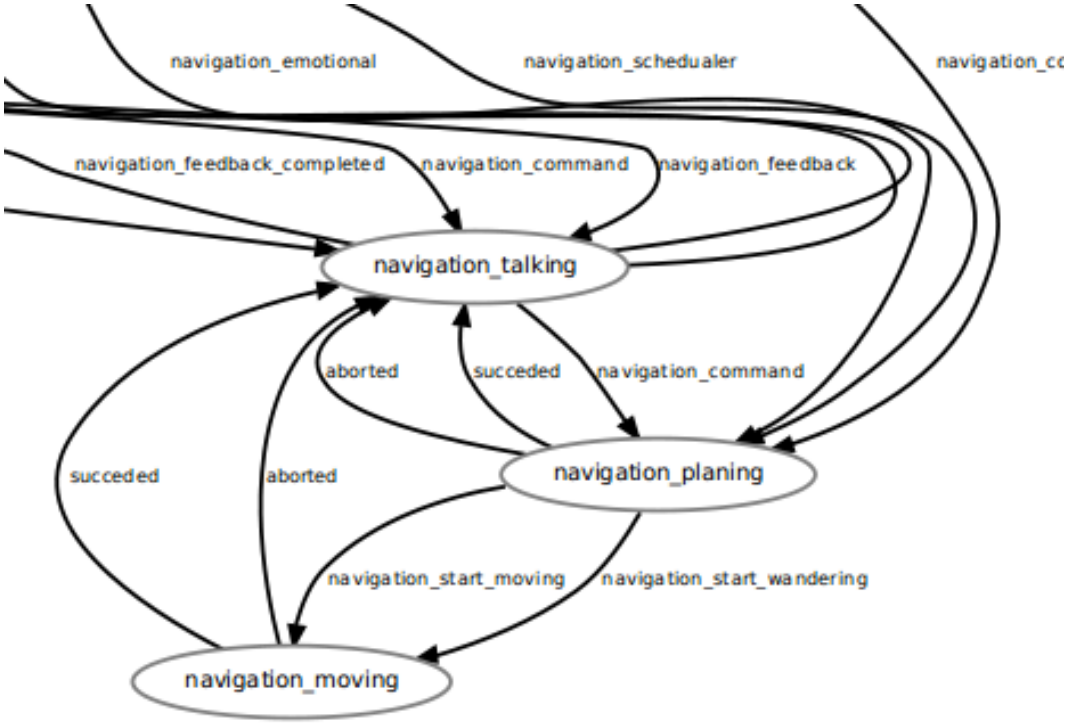

Figure 7.1: *Shows how the navigation action server are connected in the controller's state machine.*

7.5 Action Server: The Navigation Planing State

The planning action server makes plans for where the Cyborg should move to and is always active before the moving state. The decision is made based on the event that lead to the controller con-

necting to the action server and the emotional state of the Cyborg. Figure 7.2 shows a flow chart of how the planning state is selecting a location. The flow chart shows some important behavior for the Cyborg:

- If the event is a command the Cyborg selects the commanded location.
- If it was a calendar event and is angry, the Cyborg ignores the event and selects a ”non crowded” place (to be alone).
- If the event comes from the controller’s motivator and the Cyborgs emotion is happy, a ”crowded” place is selected. If the emotion is board or curious, it selects to wander.
- Feedback is given to the controllers emotion system, based on if the Cyborg is ”bossed” around by the user or not.

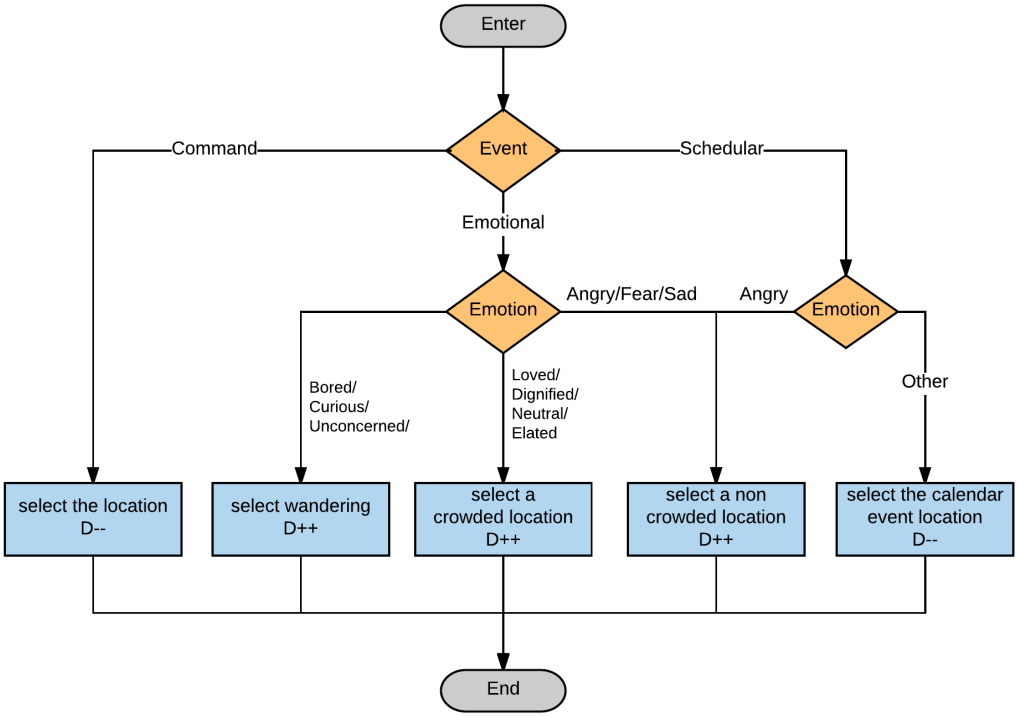

Figure 7.2: *Shows a flow chart of selecting a location in the planing state.*

7.6 Action Server: The Navigation Moving State

When the Cyborg’s controller connects to this action server, the Cyborg starts moving. The action server either moves the Cyborg to a specific location determined by the planning state or starts

wandering around. Figure 7.3 shows a flow chart of the moving state. If the Cyborg is going to move, it connects to the ROSARNL node, sends the coordinate for the location and starts moving, if it on the other hand is going to wander around, it activates the wander functionality of ROSARNL. When the Cyborg stopes wandering or moving, it will cancel any ROSARNL operation and send an event (”succeeded” or ”aborted”, depending on if there was a problem or not) to the controller.

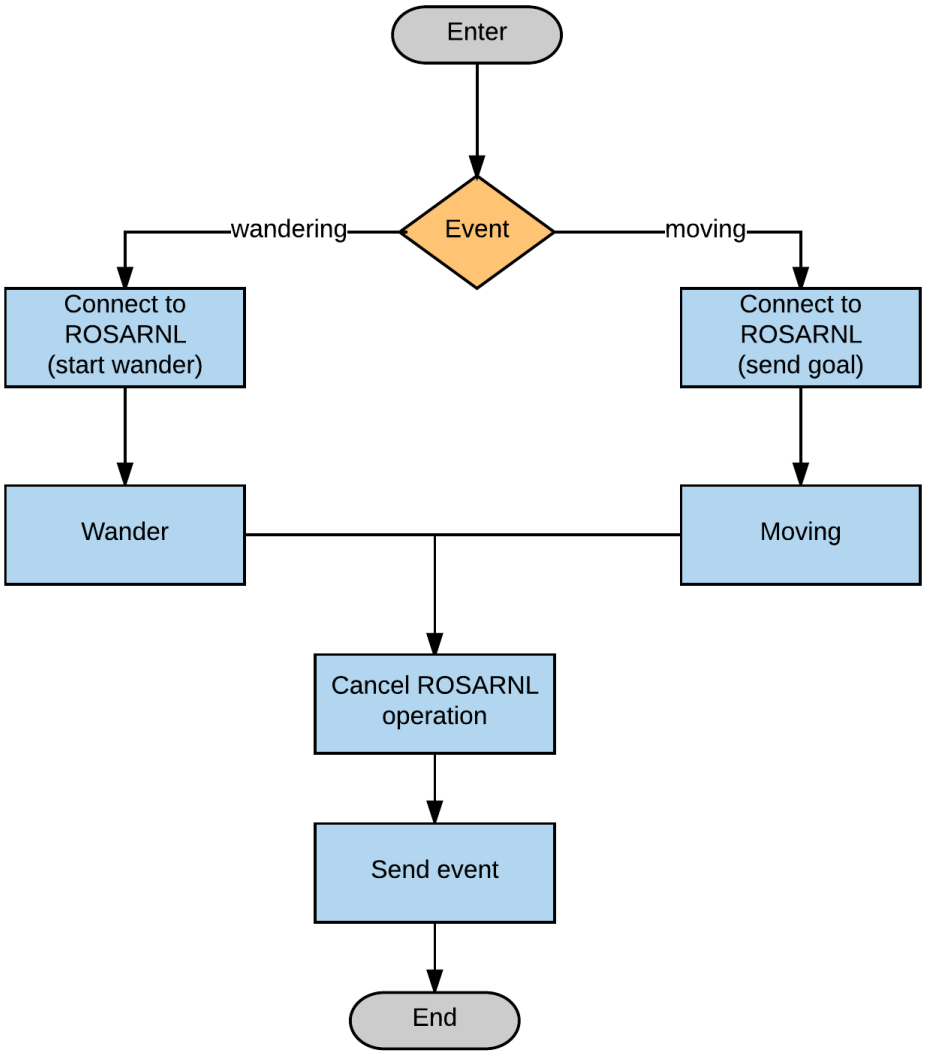

Figure 7.3: *Shows a flow chart of the moving state.*

7.7 Action Server: The Navigation Talking State

The talking server provides talking (about navigation) behavior for the Cyborg. What the Cyborg says in this state will depend on what event caused the controller to connect to the action server and the emotional state of the Cyborg. Several events may lead to this state:

- succeeded (arrived at a location): If it arrived at a location because a user asked it to show the way to a location, it will tell the name of the location, if not it will say what it thinks of the place based on the emotional state.
- aborted (unable to move to a location): The Cyborg will tell it's stuck.
- navigation_feedback: The Cyborg will say what it thinks of the place based on the emotional state.
- navigation_information: The Cyborg will ask if it should show the way.
- navigation_command: The Cyborg will ignore the command if it is angry, otherwise it will ask for confirmation.

8. Implementation of the Controller Module

This chapter describes the implementation specific details of the Controller module described in chapter 6. The implementation is done in Python and it is a single ROS node named *cyborg_controller*. The source code is delivered.

8.1 The Controller's State Machine

The state machine part of the controller is implemented using SMACH (described in chapter 2.3) as the state machine library. There are several reasons for selecting SMACH as the state machine library. At is core it is independent of ROS, which means that if in the future is decided not to use ROS, the module can still use SMAC, but at the same time it integrates very well with ROS. It comes in a ROS package. It has powerful tools (installed as separate ROS packages) that allows for live monitoring and inspection of the state machine's configuration. It also comes with premade classes that can be used to directly connect to other nodes using for example ROS services and messages, which is grate for possible changes in the future. It is well developed and it is possible to see the source code. SMACH is using the state design patter, i.e., each state is implemented as a class derived from a superclass with a interface that the state machine (SMACH) can use. In SMACH the executed code is located inside the derived class and the derived class is added to the SMACH state machine (hereafter the SMACH state machine is referred to as smach).

The states in the controller's state machines is not going to execute the behavior modules code, but instead using an action client to connect to the behavior modules action server. The state will also contain some state specific data. Since the code that the state needs to execute is the same for all states, only one derived class is needed. The derived class is derived from *smach.state* and is called *Module*. The Module is implemented in *module.py*. The Module constructor takes in a state name (should be unique), the ROS topic name where the action server is, a list of valid transitions (event name and the next state) and the list of resources that the action server need access to (for the gatekeepers). When the constructor is called it creates a publisher for state changes and a subscriber for events. The subscriber adds incoming events to a list of unhandled events.

When smach goes to a new state it calls the Module's execute method, which takes in the *userdata*. The userdata concept is described in chapter 2.3. The userdata provides the name of the

previous state, the event that lead to the this state and a list of event that not yet have been handled. When the method is called it does the following:

1. Updates the ROS parameter server with it's own resource list.
2. Publishes the state change.
3. Connects to the action server.
4. Sends the goal to the action server (tells it to execute). The goal message contains the previous state's name, the event name and the current state name.
5. Waits until the goal succeeds, the goal is aborted or there arrives an event that leads to a state change.
6. When it does, it updates the userdata with the data for the next state and
   (a) If it action server set it state as success or aborted, the execute method return the event (for smach to handle).
   (b) If it was an other event, it cancels the goal, and waits until it receives preempted form the action server. The execute method then returns the event (for smach to handle).

8.1.1 Publishing State Changes

The state machine publishes the state changes on a topic called *cyborg_controller/state_change*. The topic uses a message type of *SystemState.msg* implemented in the *cyborg_controller* node. This message contains three strings, one for the event that lead to the system state change, one for the previous state, and one for the new state. The message is given below:

```
string event
string from_system_state
string to_system_state
```

The Python code below show how to subscribe to it in a behaviour module:

```
import roslib
import rospy
from cyborg_controller.msg import SystemState

def state_callback(self, data):
```

```
    print(data.to_system_state)

rospy.init_node("node_name", True)
state_subscriber = rospy.Subscriber("cyborg_controller/state_change",
    SystemState, self.state_callback, queue_size=100)
rospy.spinn()
```

8.1.2 Registration of Events

The state machine subscribes to a topic called *cyborg_controller/register_event*, where events can be registered. The topic uses a message type of *String*. This message contains a single strings, with the event name. The Python code below show how to register an event from a behavior module:

```
import roslib
import rospy

rospy.init_node("node_name", True)
event_publisher = rospy.Publisher("cyborg_controller/register_event", String,
    queue_size=100)
event_publisher.publish("event_name")
rospy.spinn()
```

8.1.3 Adding Modules to the State Machine

How to add a Module (state) to the state machine will depend on the state, if it is a single state, a hierarchy state or a parallel state, see chapter 2.3. The simplest is a single state and can be added to the state machine by adding a single line of code in *controller.py*. The place to add the code is marked with a line marked with "ADD MORE STATES BELOW" and "STOP ADDING YOUR STATES" in the file. The line of code needed is *smach.StateMachine.add(label, state=Module(state_name, actionlib_name, transitions, resources), transitions, remapping)*. An example is given below:

```
smach.StateMachine.add(label="state_name",
    state=Module(state_name="state_name",
        actionlib_name="cyborg_module_name/actionserver",
        transitions={"aborted":"idle", "event_A":"state_A"},
        resources={"trollface":"cyborg_module_name"}),
    transitions={"aborted":"idle", "event_A":"state_A"},
```

```
    remapping=state_machine_remapping)
```

It is also important to add transitions from other states to the new state as well by editing other states transitions.

8.1.4 Using SMACH Viewer to View the State Machine at Runtime

The Python code below shows how the smach is connected to the SMACH Viewer:

```
import smach_ros
# [...] creating the smach state machine ex...
sis = smach_ros.IntrospectionServer('controler_viewer', state_machine, '/
    controler_viewer')
sis.start()
```

The state machine can then be viewed by using the SMACH Viewer:

```
rosrun smach_viewer smach_viwer.py
```

8.2 The Controller's Emotion System

The emotion system described in chapter 6.2 is implemented in *emotionsystem.py* in the *cyborg_controller* node. The possible emotional state is the states given in table 2.2. The initial state is the neutral state. The emotion system subscribes to a ROS topic where it can receive feedback from modules in the form of changes in the PAD values. When the emotion system receives feedback, it updates the current PAD values using the formulas described in the design chapter. It then uses the new values to find the closest emotional state, using the least square method, that is within a threshold. If no state is fund, it sets the emotional state to neutral. When an emotional state is selected, it publishes the emotional state.

How can Modules Influence the Emotional State?

The emotion system is subscribing to a topic called *cyborg_controller/emotional_feedback*. The topic uses a message type of *EmotionalFeedback.msg* implemented in the *cyborg_controller* node. The message contains three float values. One for each of the PAD values. The values represent the change in the PAD values and can be negative for a decreasing value, zero for no change and positive for an increasing value. The changes will be added to the current PAD values. The message is given below:

```
float32 delta_pleasure # Changes in pleasure
float32 delta_arousal # Changes in arousal
float32 delta_dominance #Changes in dominance
```

The Python code below show how to provide feedback from a behavior module:

```
import roslib
import rospy
from cyborg_controller.msg import EmotionalFeedback

rospy.init_node("node_name", True)
emotion_publisher = rospy.Publisher("cyborg_controller/emotional_feedback",
    EmotionalFeedback, queue_size=100)
msg = EmotionalFeedback()
msg.delta_pleasure = -0.01
msg.delta_arousal = 0.10
msg.delta_dominance = 0.00
emotion_publisher.publish(msg)
```

How Can Modules Get the Emotional State?

If a module gives feedback to the emotion handler, in form of changes in the PAD values and these values results in a change in the emotional state, the emotional handler will publish the new emotional state on a topic called *cyborg_controller/emotional_state*. The topic uses a message type of *EmotionalState.msg* implemented in the *cyborg_controller* node. The message contains the previous state, the new current state and the current PAD values:

```
float32 from_emotional_state
float32 to_emotional_state
float32 current_pleasure
float32 current_arousal
float32 current_dominance
```

The Python code below show how behavior modules can subscribe to the emotional state:

```
import roslib
import rospy
from cyborg_state_machine.msg import EmotionalState

current_emotion = "neutral"
rospy.init_node('listener', True)
```

```
7
8 # Updates the current emotion when the emotion subscriber receives data
9 def emotion_callback(self, data):
10     global current_emotion
11         current_emotion = data.to_emotional_state
12
13 emotion_subscriber = rospy.Subscriber("cyborg_controller/emotional_state",
    EmotionalState, emotion_callback)
14 rospy.spin()
```

A behavior module can also get the current state by using a service at *cyborg_controller/get_emotional_state*. This service uses a service type *EmotionalStateService.srv* defined in the *cyborg_controller* node.

8.3 The Controller's Motivator

The motivator described in chapter 6.3 is implemented in *motivator.py* in the *cyborg_controller* node. All motivational events is stored in a database. The motivator is subscribing to the state change topic, when it receives message about a state change it updates list of possible event it can select. The motivator is also subscribing to the emotional state topic, and when a message arrives, it updates the current PAD values.

When the motivator finds an event that has a larger reward then the cost associated with event, it publishes the event on the register event topic and it publish the reward on the emotional feedback topic. It also adds 0.33 to the event drive for the selected event and decreases the event drive for other drives with 0.05. The formulas used, is the formulas described in chapter 6.3.

8.3.1 Adding Motivational Events to the Motivator

Adding an emotional event to the motivator require one line of code in the controller.py file:

```
1 database_handler.add_event(state="state_name", event="event_name",
    reward_pleasure=0.10, reward_arousal=0.00, reward_dominance=-0.05,
    event_cost=0.33)
```

Alternatively, the same information can be added to the SQLite database that the motivator uses. This can for example be done through a graphical program like *sqlite browser*. The event added to the database must also exist in the state machine as a valid transition from the state.

9. Implementation of Behaviour Modules

There are several things that must be done implementing a behavior module and somethings that are optional. This chapter provides a brief summery of what must be done. A behavior module must implement action servers (see chapter 2.1.1). The controller is made aware of an action server by adding it as a state in the controllers state machine (see chapter 8.1). The behavior module's action servers becomes active when the state machine enters the state. The state machine is event driven, and the behavior modules can detect events and send these to the state machine (as described in chapter 8.1) or it can add events to the motivator as described in chapter 8.3. A behavior module can use the emotion system to get the Cyborg's internal emotion. How to get the emotional state is described in chapter 8.2 and a module can send emotional feedback as described in chapter 8.2. The following sections describes the implementation of an Idle module and a Navigation Module.

9.1 Implementation of the Idle Module

The Idle module is a behavior module that adds behavior to the Cyborg, however as the name implies, it doesn't add much behavior, simply being idle. When the Idle module is active, it will decrease the PAD values, to simulate boredom. The initial state in the controller's state machine is the idle state, therefor the idle state is a required state. The idle state is implemented in the Idle module. The Idle module is implemented in C++ in *idle.cpp* in the *cyborg_idle* node. It has one action server for the idle state and the state is added to the state machine:

```
1 idle_transitions = {"conversation_interest":"conversation", "
    navigation_schedualer":"navigation_planing", "navigation_emotional":"
    navigation_planing", "aborted":"idle", "navigation_command":"
    navigation_planing", "music_play":"music"}
2 idle_resources = {} # Idle does not require any resources
3 smach.StateMachine.add(label="idle", state=Module(state_name="idle",
    actionlib_name="cyborg_idle/idle", transitions=idle_transitions, resources
    =idle_resources), transitions=idle_transitions, remapping=
    state_machine_remapping)
```

The complete source code for the idle state is delivered.

9.2 Implementation of the Navigation Module

This chapter describes the implementation specific part of the Navigation module designed in chapter 7. The Navigation module is written in Python. The implementation uses ROS and creates a node called *cyborg_navigation*. A map of the hallways of Glassgården at NTNU was made using the method described in chapter 2.7.1. The resulting map is shown in figure 9.1. The complete source code is delivered.

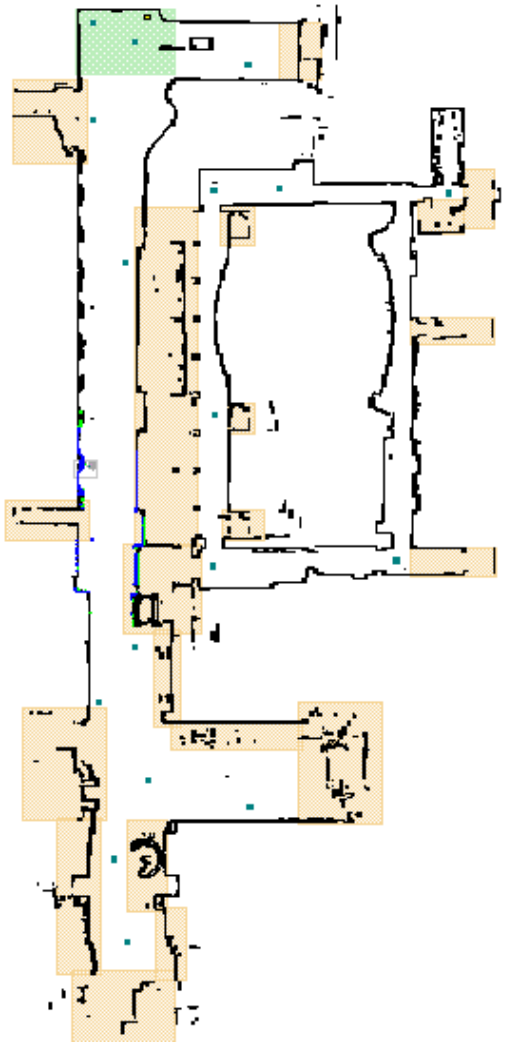

Figure 9.1: The map over Glassgarden, NTNU.

9.2.1 The Database: Content of the Database and How to Add More Data to the Database?

The database is created using SQLite which comes included in Python. The Navigation module has a database handler class that can be used to query, update and add data to the database. The Navigation modules uses this handler class to get locations (and their coordinates), times of scheduled events and expressions that the Cyborg can use. If the Navigation module does not detect a database, a new database will be created and some default data will be added automaticly. Currently this is data relevant for Glassgården and some default expressions. More data can be added to the database either by using the database handler class or by using a graphical tool like *SQLite Browser*.

9.2.2 The Planing, Moving and Talking State Implemented as Action Servers

The navigation modules offers three states (planning, moving and talking) for the controllers state machine. These three states is implemented in the in the navigation module as action servers. The controller can then connect to these servers over the actionlib protocol. The servers are using the ROS topic name *cyborg_navigation/planing*, *cyborg_navigation/moving* and *cyborg_navigation/talking*.The servers are created and started when the navigation node starts up:

```
1 self.server_planing = actionlib.SimpleActionServer(rospy.get_name() + "/
    planing", StateMachineAction, execute_cb=self.server_planing_callback,
    auto_start = False)
2  self.server_moving = actionlib.SimpleActionServer(rospy.get_name() + "/
    moving", StateMachineAction, execute_cb=self.server_moving_callback,
    auto_start = False)
3  self.server_talking = actionlib.SimpleActionServer(rospy.get_name() + "/
    talking", StateMachineAction, execute_cb=self.server_talking_callback,
    auto_start = False)
4  self.server_planing.start()
5  self.server_moving.start()
6  self.server_talking.start()
```

The code to execute for each state is implemented inside the action servers respective callback functions. The navigation states uses the design described in the design chapter **??**. An important part of the controller's design is that states must be able to preempt each other. Therefor the states checks if they are preempted by the controller:

```
# Chek if controller has preemted the state
if self.server_moving.is_preempt_requested():
  # Do things needed to be done, before leaving state
  self.server_moving.set_preempted()
```

9.2.3 Receiving Data from Other Modules, Detecting of Events from Texts and Scheduled Events

The navigation module subscribes to the speech to text topic:

```
self.text_subscriber = rospy.Subscriber("/text_from_speech", String, self.
    text_callback, queue_size=100)
```

When data (text) is received on the subscriber, the callback function text_callback() is called. This callback function searches for relevant keywords and locations that it has in its database. If it find a mach, it send the related event to the controller:

```
self.event_publisher = rospy.Publisher("/cyborg_controller/register_event",
    String, queue_size=100)
self.event_publisher.publish("navigation_event")
```

The possible events are *navigation_command*, *navigation_information* and *navigation_feedback* (if the user ask for the Cyborg's opinion about a place). A scheduler checks the database for ongoing scheduled event, it there is, a *navigation_schedualer* event, is sent to the controller, just like the other events. The Navigation module depends on the Cyborg's emotional state. It receives the Cyborg's emotion from the controller by subscribing to the emotion topic. When it receives the Cyborg's emotion, a callback function stores the emotional state.

```
self.emotion_subscriber = rospy.Subscriber("/cyborg_controller/emotional_state
    ", EmotionalState, self.emotion_callback, queue_size=100)

# Updates the current emotion when the emotion subscriber recives data from
    the controller (emotion system)
  def emotion_callback(self, data):
    self.current_emotion = data.to_emotional_state
```

It also sends feedback for the emotion system, e.g., when it moves, to the controller, so it has an emotion publisher witch it uses for publishing feedback to the controller:

```
self.emotion_publisher = rospy.Publisher("/cyborg_controller/
    emotional_feedback", EmotionalFeedback, queue_size=100)
```

```
self.send_emotion(pleasure=0.00, arousal=0.00, dominance=0.05)
```

9.2.4 Connecting to the ROSARNL Node for Controlling the Robot Base

The Navigation Module is controlling the robot base through ROSARNL. As explained in chapter 2.7, the ROSARNL is making navigation available through an actionlib server. The navigation module has an actionlib client that it uses:

```
self.client_base = actionlib.SimpleActionClient("/rosarnl_node/move_base",
    MoveBaseAction)
pose = geometry_msgs.msg.Pose()
# [...] The data is then filled in to a geometry message
goal = MoveBaseGoal()
# [...] The data is then entered into the goal message
self.client_base.send_goal(goal, self.client_base_done_callback, self.
    client_base_active_callback, self.client_base_feedback_callback)
```

The callback function added in the last line, *self.client_base_done_callback*, is called when the base reaches it destination or if the base is unable to get to the destination. When that happen the navigation server will set the actionlib state it has with the controller to eather succedded or aborted, respectivly. It the navigation module wants to start wandering. It conntacts the robot base through ROSARNL. It uses a ROS service call. The service call is:

```
from std_srvs.srv import Empty
baseStartWandering = rospy.ServiceProxy("/rosarnl_node/wander", Empty)
baseStartWandering()
```

When the Navigation Module wants to stop, it uses a cancel method to stop the robot base.

9.2.5 Integrating the Navigation Module with the Controller

The navigation module has three action servers that the controller can use. These action servers must therefor be added to the controllers state machine as three separate states. The navigation modules is added as follow:

```
# Open the container
with state_machine:
smach.StateMachine.add(label="navigation_moving", state=Module(state_name="
    navigation_moving", actionlib_name="cyborg_navigation/moving", transitions
    = {"aborted":"navigation_talking", "succeded":"navigation_talking"} ,
```

```
    resources={"base":"cyborg_navigation"}), transitions= {"aborted":"
    navigation_talking", "succeded":"navigation_talking"} , remapping=
    state_machine_remapping)
```

The navigation module has an event for the motivator, *navigation_emotional*. If the motivator selects the event, the Cyborg will move around on its own. The event is added to the motivator using the database handler:

```
database_handler.add_event(state="idle", event="navigation_emotional",
    reward_pleasure=0.00, reward_arousal=0.05, reward_dominance=-0.01,
    event_cost=event_cost)
```

10. Proof of Concept

10.1 Setup and Configuration of a System for Testing the Controller Module and the Navigation Module)

A proof of concept system was created for testing the controller and the navigation behaviour module. In order to be able to test the controller a bit more then with just one module, some additional (small/test) modules was developed:

- A text to speech module was need (since the Trollface currently not in use).
- The voice input module is currently not used. A simple command line tool was made for testing purposes. The tool allows for testing the controller by sending events to the state machine, sending text input (one the same topic as the STT would send on) and manipulation of the emotion system (setting emotions and turning it on and off). This tool is strictly meant for testing/debugging purposes, it is delivered incase future developers may have use for it.
- Some states for the controller to interact with was needed. These are simple placeholder states, until more modules are integrated into the controller. The modules offer the following states:
  - Music state: Starts to play some music.
  - Conversation state: if text input is detected it enters a conversation state.
  - Joke state: The Cyborg asks the user if he know any jokes.
  - Selfie state: The Cyborg asks if the user want to take a selfie.
  - Follower state: The Cyborg ask if the robot should follow the user.
  - Simon Says: The Cyborg ask if the user wants to play Simon says.
  - Weather state: The Cyborg ask what the user thinks about the weather.

The states described above was added to the controllers state machine, as well as appropriate events to create a transition between the states. The controllers state machine is shown in figure 10.1. Some events (with rewards) are also added to the controllers motivator database. The database is included in the delivered files. The testing of the navigation module is done using the robot base simulator (provided by the manufacturer). Instead of the ROSARNL node connecting to the robot base, it connects to the simulator. The map of Glassgården NTNU is used, along with

a database of some locations. The database is given some dates and times of when the Cyborg should be at a certain locations. The test system has some requirements (dependencies). All the dependencies are listen in appendix A.

The source code must be placed in the catkin workspace's src folder and built using the catkin_make tool. The system was run by first starting the MobileSim software and then starting *roscore* and the ROSARNL node (*rosrun rosarnl rosarnl_node*). The controller along with all the other modules can be run using the launch file:

```
1 roslaunch cyborg_controller controller.launch
```

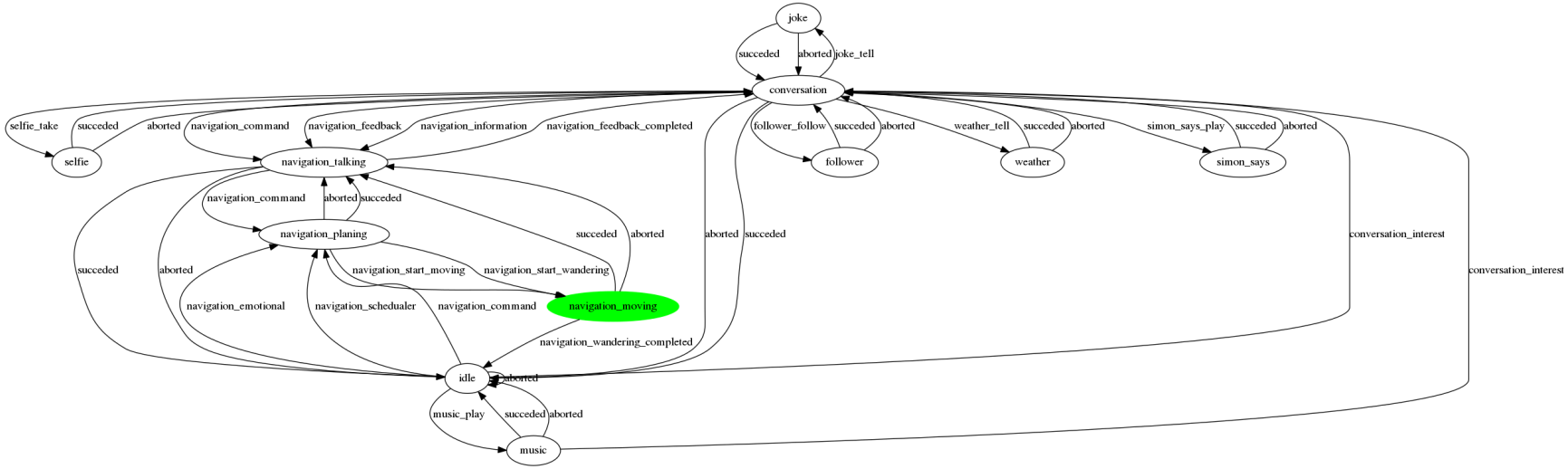

Figure 10.1: *Shows an overview of the controller's state machine. This is an image generated by the state machine monitorer module. The currently active module is marked with green.*

10.2 Observations Made During Testing

When testing the system described above, there was made several observations of how the system behaved. This section will briefly go through some of the important observations made of the system during testing. The following syntax is describing a state transition:

```
current_state - event_that_lead_to_transition - next_state
```

User input is done as text input through the command line tool (the tool publishes the text on the same ROS topic as the speech to text is publishing on). In the observations, the sentences spoken by the Cyborg is the voice from the TTS module. The system is using the MobileSim software to simulate the robot base hardware. The observations about the Cyborg moving is done in the MobileSim/MobileEyes software.

10.2.1 Scenario A: A User Interact With the Cyborg

Observation 0: Startup

```
When the system is started, the first things observed is that
the controller's state machine is showing that the idle state is active
and the emotional state is neutral.
The Cyborg is located at the information desk.
```

Observation 1: The Cyborg's First Action

```
[Cyborg is unconcerned]:
idle - navigation_emotional - navigation_planning
navigation_planning - navigation_start_wandering - navigation_moving
[the Cyborg starts to wander]
[the Cyborg wanders in about 120 secounds]
[Cyborg is elated]
navigation_moving - navigation_wandering_completed - idle
```

Observation 2: The Cyborg's Second Action

```
idle - music_play - music
[Cyborg is curius]
[the Cyborg is playing music]
music - succeeded - idle
```

Observation 3: User Asks the Cyborg's question

```
User: "hello"
idle – conversation_detected – conversation
User: "What do you think of el6?"
conversation – navigation_feedback – navigation_talk
Cyborg [Cyborg is curius]: "What is this place? A el6 you say?"
navigation_talk – navigation_feedback_completed – conversation

User: "Where is the cafeteria ?"
conversation – navigation_information – navigation_talk
Cyborg: "I think I know where that is, would you like me to show you?"
User: "yes"
Cyborg: "At once."
navigation_talk – navigation_command – navigation_planing
navigation_planing – navigation_start_moving – navigation_moving
[Cyborg is moving to words the cafeteria]
[Cyborg arrives at the cafeteria]
navigation_moving – navigation_feedback – navigation_talking
Cyborg: "Human, this is cafeteria"
navigation_talking – succeeded – idle

User: "go to entrance 2"
idle – navigation_command – navigation_planning
navigation_planning – navigation_start_moving – navigation_moving
[the Cyborg moves from cafeteria towards the entrance 2]
[the Cyborg arrives at entrance 2]
navigation_moving – succeeded – navigation_talking
Cyborg [Cyborg is curius]: "What is this place? A entrance 2 you say?"
navigation_talking – succeeded – idle
```

10.2.2 Scenario B: The Cyborg

Observation 0: Startup

```
When the system is started, the first things observed is that
the controller's state machine is showing that the idle state is active
and the emotional state is neutral.
The Cyborg is located at the information desk.
```

Observation 1: The Cyborg's First Action

```
[Cyborg is unconcerned]:
```

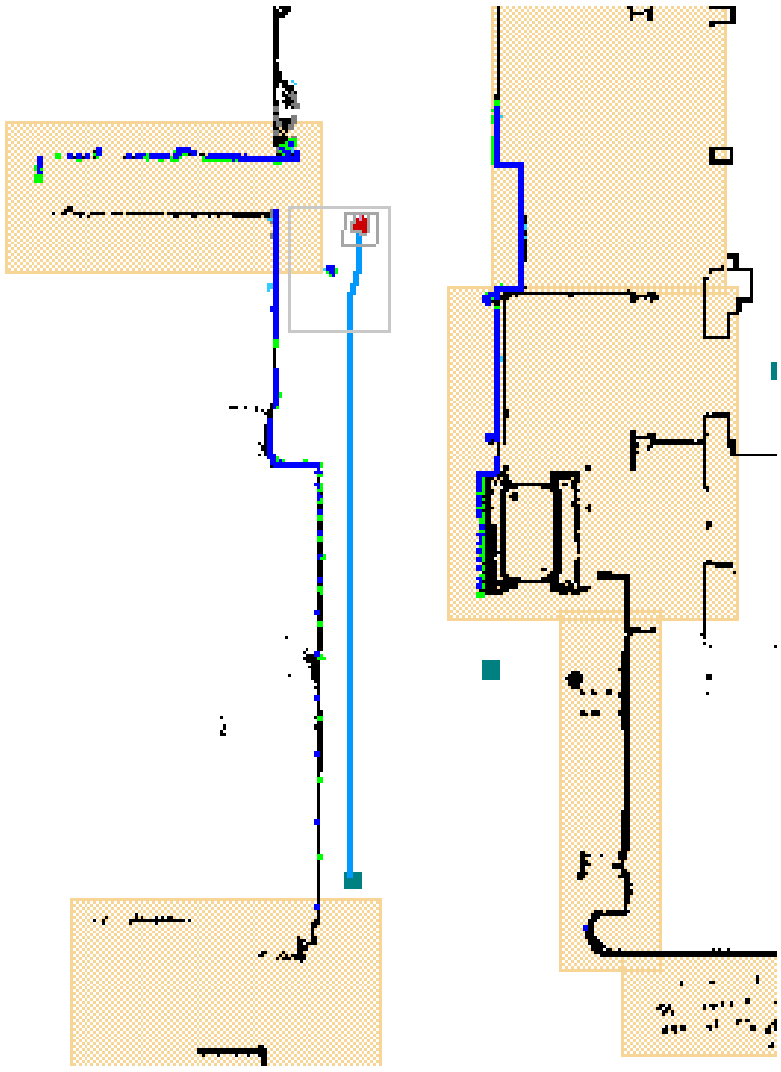

Figure 10.2: *Shows the Cyborg moving towards the cafeteria.*

```
idle - navigation_emotional - navigation_planning
navigation_planning - navigation_start_wandering - navigation_moving
[the Cyborg starts to wander]
[the Cyborg wanders in about 120 secounds]
[Cyborg is elated]
navigation_moving - navigation_wandering_completed - idle
```

Observation 2: The Cyborg's Second Action

```
idle - music_play - music
[the Cyborg is playing music]
music - succeeded - idle
```

Observation 3: The Cyborg is Curious and Wander More

```
[Cyborg is curious]
idle - navigation_emotional - navigation_planning
navigation_planning - navigation_start_wandering - navigation_moving
[the Cyborg starts to wander]
[...]
[Cyborg is elated]
navigation_moving - navigation_wandering_completed - idle

idle - music_play - music
[the Cyborg is playing music]
music - succeeded - idle

[Cyborg is curious]:
idle - navigation_emotional - navigation_planning
navigation_planning - navigation_start_wandering - navigation_moving
[the Cyborg starts to wander]
[Cyborg is elated]
navigation_moving - navigation_wandering_completed - idle

idle - music_play - music
[the Cyborg is playing music]
music - succeeded - idle

[Cyborg is curius]:
idle - navigation_emotional - navigation_planning
navigation_planning - navigation_start_wandering - navigation_moving
[the Cyborg starts to wander]
[the Cyborg wanders in about 270 secounds]
[Cyborg is elated]
navigation_moving - navigation_wandering_completed - idle
```

Observation 4: The Cyborg's Plays Music and Is Interrupted by a User

```
idle - music_play - music
[the Cyborg is playing music]
User: "Hello"
[music stoped]
music - conversation_detected - conversation
User: "Where is e16 ?"
conversation - navigation_information - navigation_talk
```

```
Cyborg: "I think I know where that is, would you like me to show you?"
User: "yes"
Cyborg: "At once."
navigation_talk - navigation_command - navigation_planing
navigation_planing - navigation_start_moving - navigation_moving
[Cyborg is moving to words el6]
[Cyborg arrives at el6]
navigation_moving - navigation_feedback - navigation_talking
Cyborg: "Human, this is el6"
navigation_talking - succeeded - idle
```

10.2.3 Scenario C: The Cyborg With a Scheduled Time and Location

Observation 0: Startup

```
When the system is started, the first things observed is that
the controller's state machine is showing that the idle state is active
and the emotional state is neutral.
The Cyborg is located at the information desk.
The Cyborg is scheduled to be at the cafeteria between 15:00 and 15:59.
The Current time is 15:29.
```

Observation 1: The Cyborg's First Action

```
idle - navigation_scheduled - navigation_planning
navigation_planning - navigation_start_moving - navigation_moving
[the Cyborg moves from information desk towards the cafeteria]
[the Cyborg arrives at the cafeteria]
navigation_moving - succeeded - navigation_talking
Cyborg [Cyborg is neutral]: "I am happy about cafeteria"
navigation_talking - succeeded - idle
```

Observation 2: A User Asking for el6

```
idle - music_play - music
[the Cyborg is playing music]

User: "hello"
[music stoped]
music - conversation_detected - conversation
User: "where is the el6"
conversation - navigation_information - navigation_talk
```

```
Cyborg: "I think I know where that is, would you like me to show you?"
User: "yes"
Cyborg: "At once."
navigation_talk - navigation_command - navigation_planing
navigation_planing - navigation_start_moving - navigation_moving
[Cyborg is moving to words the el6]
[Cyborg arrives at the el6]
navigation_moving - navigation_feedback - navigation_talking
Cyborg: "Human, this is el6"
navigation_talking - succeeded - idle

idle - navigation_scheduled - navigation_planning
navigation_planning - navigation_start_moving - navigation_moving
[the Cyborg moves from information desk towards the cafeteria]
[the Cyborg arrives at the cafeteria]
navigation_moving - succeeded - navigation_talking
Cyborg [Cyborg is neutral]: "I am happy about cafeteria"
navigation_talking - succeeded - idle
```

10.2.4 Scenario D: Conversation

Observation 0: Startup

```
When the system is started, the first things observed is that
the controller's state machine is showing that the idle state is active
and the emotional state is neutral.
The Cyborg is located at the information desk.
```

Observation 1: The Cyborg's First Action

```
[Cyborg is unconcerned]:
User: "Hello"
idle - conversation_interest - conversation

conversation - simon_says_play - simon_says
Cyborg: "Do you want to play Simon says?"
simon_says - succeeded - idle

conversation - joke_tell - joke
Cyborg: "Do you know any funny jokes?"
joke - succeeded - idle
```

```
12
13 conversation - weather_tell - weather
14 Cyborg: "What do you think of the weather?"
15 weather - succeeded - idle
16
17 conversation - selfie_take - selfie
18 Cyborg: "Do you want to take a selfie?"
19 selfie - succeeded - idle
20
21 conversation - simon_says_play - simon_says
22 Cyborg: "Do you want to play Simon says?"
23 simon_says - succeeded - idle
24
25 conversation - weather_tell - weather
26 Cyborg: "What do you think of the weather?"
27 weather - succeeded - idle
28
29 conversation - follower_follow - follower
30 Cyborg: "Can I follow you?"
31 follower - succeeded - idle
32
33 conversation - joke_tell - joke
34 Cyborg: "Do you know any funny jokes?"
35 joke - succeeded - idle
36
37 conversation - selfie_take - selfie
38 Cyborg: "Do you want to take a selfie?"
39 selfie - succeeded - idle
40
41 conversation - weather_tell - weather
42 Cyborg: "What do you think of the weather?"
43 weather - succeeded - idle
44
45 [Cyborg is unconcerned]
```

11. Discussion

11.1 Does the Implementation of the Controller Module Satisfy the Specifications?

In chapter 4, specifications for a controller module was determined. The specification, lead to a design for the controller module and an implementation. Table 11.1 lists all specifications for the controller, a short description, a comment for each specification and a column for checkmarks (X means satisfied). The table has all rows checked. It becomes clear form the table that the implementation of the controller satisfies the specifications.

11.2 Does the Implementation of the Navigation Module Satisfy the Specifications?

In chapter 5, specifications for a navigation module was determined. The specification, lead to a design for the navigation module and an implementation. Table 11.2 lists all specifications for the navigation module, a short description, a comment for each specification and a column for checkmarks (X means satisfied). The table has all rows checked. It becomes clear form the table that the implementation of the controller satisfies the specifications.

Spec	Description	Comment	Check
1	Implemented in ROS.	The controller is implemented in a single ROS node.	X
1.1	Modules as separate ROS nodes.	The behaviour is implemented as separate ROS nodes.	X
2	Consistency.	Solves this by using a state machine.	X
2.1	Detection of events in modules.	It is the modules that detects the events.	X
2.2	Controller decides.	Through a state machine and actionlib protocol.	X
2.3	Preemption.	Preemption is possible.	X
2.4	Controller control gatekeepers.	The controller controls the gatekeepers.	X
2.5	Updates.	Subjective and relative, but good enough.	X
3	Optional: Graphical viewer.	A limited graph view method was developed.	X
4	Optional: Gatekeeper.	Implemented by Kvello [36].	X
5	Emotion System.	Exists.	X
5.1	Feedback from modules.	Accepts feedback.	X
5.2	Setting an emotional state.	Selects an emotional state.	X
5.3	Publishing the emotional state.	Publishes the state.	X
6	Motivator.	Exist.	X
6.1	Uses events.	Publishes events for changing the state.	X
6.2	Aware of active state.	Subscribes to the state change topic.	X
6.3	The motivator only motivates.	Behaviour in separate modules.	X
6.4	Reward system.	Uses a reward system.	X

Table 11.1: *Evaluation of the implementation of the controller against the specifications.*

Spec	Description	Comment	Check
1	Use ROSARNL node.	The Navigation Module is using the ROSARNL Node to communicate with the robot base.	X
2	May use map.	It is using a map of Glassgarden, NTNU	X
3	Database of known locations.	It has a database of location on the map.	X
4	Using emotions.	It provides emotional feedback for example when it is commanded to new locations and when it is moving. The result from the planing state depends on the emotional state of the robot.	X
5	Desirable features.	It has some features relevant for short term goal.	X
5.1	Directions.	The user can ask where something is and the robot asks if it should show the way if it knows where it is.	X
5.2	Command the robot.	The user can tell the robot to go to a location.	X
5.3	Scheduling events.	It is possible to scedual events.	X
5.4	Add event to the motivator.	The motivator can use an event to let the robot move around.	X

Table 11.2: *Evaluation of the implementation of the navigation module against the specifications.*

11.3 What is Happening in the Observations Made in the Proof of Concept? Detailed Descriptions and Explanations

The start up observation (observation 0) made in all scenarios is that when the controller's state machine starts up it sets the idle state, from the idle module, as the active state. When the state machine enters the idle state it connects to the idle module by using the actionlib protocol. The idle module action server is activated and starts to execute. This is correct behavior. It is important that the system is in a well defined state when it starts up. It is also observed that the selected emotion is neutral.

11.3.1 Explanation of Scenario A

In observation 1, the Cyborg wanders for 120 seconds. The reason that the Cyborg starts to wander is because the state machine receives a navigation_emotional event from the motivator. Shortly after startup, the Cyborg gets unconcerned (because the idle state decreases the PAD values). The motivator finds the navigation_emotional event to be the event it can select to get the larges reward (increase in the PAD values). When the state machine receives the event it preempts the idle module, and enters the navigation_planning state and connects to the actionlib server for that state. Since the Cyborg is unconcerned, the planning states selects that the Cyborg should start to wander, and a wandering event is sent to the controller. The event leads to the navigation_moving state and the Cyborg starts to wander. The Cyborg is set to stop to wander when it is no longer in the unconcerned, board or curios state, so when the Cyborg becomes elated, the module sent an event to the state machine telling it to leave the wandering state.

In observation 2, the Cyborg is entering the music states and turns on some music. When the music is done playing it goes back to idle.

In observation 3, a user says hello and the Cyborg enters the conversation state. The user then asks what the Cyborg thinks of el6 (an auditorium). The navigation module receives the text input and since the phrase passes the keyword searches, it sends an event to the state machine asking for control. The controller grants control to the navigation_talking state and the cyborg replays. The replay is as it is because the Cyborg is in curious emotional state. Then the Cyborg return to the conversation state. The user then asks where the cafeteria is and in a similar manner, the cyborg replays if it should show the way. The user says yes and the Cyborg then drives to the cafeteria. It is important to point out that the cyborg is not aware of where the user is, it simply moves to

the cafeteria (so it is the users responsible for keeping up etc..). When the cyborg arrives at the cafeteria it says that this is the cafeteria. The user then tells the Cyborg to go to entrance 2, which it does.

11.3.2 Explanation of Scenario B

The Cyborg is changing between walking around and playing to music, until a user comes and asks for el6. The Cyborg knows where it is and shows the way.

11.3.3 Explanation of Scenario C

- The Cyborg is scheduled to be at the cafeteria, so it goes there.
- A user ask the Cyborg where el6 is, and the Cyborg goes to el6.
- The Cyborg goes back to the cafeteria.

This shows that the scheduler is working. The scheduler does not prevent the Cyborg from moving around, but it will return to the given location, as shown in the scenario.

11.3.4 Explanation of Scenario D

- A user says hello and the Cyborg goes to the conversation state.
- The user is not asking any questions, so the Cyborg asks some questions.
- The Cyborg's emotion does not change much. It remains unconcerned.

This scenario shows that the motivator part of the controller is creating events that causes the Cyborg to ask questions when the user is not saying anything. Of course the motivational events is primarily intended as events leading out from the idle state and is not intended to be used for asking a single question, however the selected questions are related to modules that does exist and it would be possible to integrate them with their respective modules.

The reason for creating more module (states) for this test was to see how it would go with the motivator. The observation shows that the selected question is rotated nicely, i.e., it doesn't ask the same question in row. It can of course asking the same user the same question, but since there in this test has a low limit in the number of modules to select from it was to possible options, to ask the same questions or ask none. In this example it was preferable that the cyborg repeated itself, rather then doing noting. This can be adjusted by setting the cost value higher etc.

Spec	Description	Comment	Check
Controller			
2	Consistency.	When inspecting the active state it is shown that the Cyborg is always in a well defined state. There is no observations that is indicating otherwise. The state machine seams to be working.	X
2.3	Preemption.	Observation B.4 (Cyborg plays music, interrupted by user) shows the preemption is possible.	X
3	Graphical Viewer.	SMACH Viewer is displaying a graph.	X
5	Emotion System.	The Cyborg is changing emotion.	X
6	Motivator.	The Cyborg is doing things on its own.	X
Navigation Module			
2	May use map.	There is a viewable map in the simulator.	X
3	Database of known location.	The Cyborg can show the user some locations.	X
4	Using emotions.	The Cyborg is doing things based on the emotional state it is in.	X
5	Desirable features.		X
5.1	Directions.	Observation A.3: User asking for cafeteria.	X
5.2	Command the Cyborg.	Observation A.3: User commanding the Cyborg.	X
5.3	Scheduling events.	Observation C.1	X
5.4	Add events to the motivator.	The Cyborg is wandering because of events from the motivator.	X

Table 11.3: *Observations Matching the Expected Behavior Based on the Specifications*

11.4 Does the Observations Match the Expected Behavior Based on the Specifications

Some of the specification for the controller and the navigation module, should be directly observable. Does the observations made match the expected behavior based on the specification? Table 11.3 lists all the specification for the controller and the navigation module that should be (somewhat) observable in the Cyborgs behavior or through tools. The table contains a short description, a comment for each specification and a column for checkmarks (X means marched). The table has all rows checked. It becomes clear form the table that the observation marches the expected behavior based on the specifications.

11.5 Discussion About the Selected Methods, Models and Protocols for the Controller

The controller is using the actionlib protocol for activating the modules. The testing and observations shows that this is working without problems: The controller can activate and preempt modules. The decision to select the ROS actionlib protocol was therefore a good choose. Furthermore the subscribers and publishers seams to be working as intended and there is not observed anything indicating that this does not work as intended.

The modules (actionlib servers) are organized into a state machine so that the Cyborg always becomes in a well defined state preventing multiple conflicting modules to run at the same time. The test system uses only regular single states and no parallel and hierarchy states, however the selected state machine library does have these features, which will make it possible to do so in the future when more modules get connected to the controller. Using an event driven state machine was probably a good idea and SMACH was probably a good choose as a library.

The PAD model was selected to be used for the Cyborg's emotions. The controller receives feedback from other modules and publishes the updated emotion. The observations shows that this is working, in the sense that the answers from the navigation module, when user asking, is a curios answer when the Cyborg is in the curios state and that Cyborg selects to wander when the motivator tells it to move and stops to wander when it is elated. It's of curse important to point out that it is only a model. In it's current form the controller is receiving the feedback from modules, however an alternative could be to get the response from the state machine, e.g., when the state machine is in a moving state. This could make it simpler for module developer, on the other hand it would place a limit on future modules and how they could use the feedback system to influence the Cyborg's mood. An alternative could be to use a combination of both. However, as it is, the system seams to be working, but it is important to select feedback values with care. It is also worth pointing out that the implemented system does support the setting of emotional state directly if it is desirable, e.g., setting it directly to angry etc.

The motivator is based on the selection of greedy reward for the drives. It only selects the action that gives the largest reward right now, it does not plan ahead or take into the account that some actions take longer time then others. The proof of concept does show that it prevents the Cyborg from remaining idle, as it was supposed to do. The selection of reward values as well as cost values are important and can change the Cyborg's behavior. It is important to select values that result in desired behavior. There are values that can be change to change what actions get selected.

If it desirable that the Cyborg only do one event over a long time, the event saturation can be sat very high. This gives some flexibility for future development. The three main drives was based on the PAD values. This seams to work just fine based on the observations. The thing about using drives it that there can be added more drives if needed and the priority functions can be changed if needed. One drive that may be interesting to use in the future, is a drive for the battery.

What about the gatekeepers? The gatekeepers is not used in the test system, because they where not needed in this particular case. If the gatekeepers are needed or not will depend on the rest of the system. The gatekeeper was designed for systems where several modules continuously talk to the output modules without any coordination from the controller. In a system where all behavior modules are connected to the controller and only publishes to the output modules when they are activated by the controller, the gatekeeper may not be needed, however they do provide a good extra safety feature as well as they can be used with modules that does not complie with the actionlib protocol or as an additional security. Since not all existing modules for the Cyborg is using action servers yet (the plan is to change it), the gatekeeper could be used until that happens.

11.6 Future Work and Can the Proof of Concept Be Used as It Is on the Cyborg?

The proof of concept system should theoretically be able to run as it is on the Cyborg (robot base) provided that all the dependencies in appendix A was installed. However, there are several issues. The Cyborg is currently without speakers (some students are currently working on it), meaning there would be no voice outputs. The Cyborg is currently without the STT module and a microphone (some students are currently working on it). This could be solved by using a keyboard and a screen, but using a screen is currently not a good solution (as the screen needs power from the wall socket). The Cyborg could receive text input over the internet, but there are connection issues when the Cyborg changes networks (moving around). All these issues are work in progress by other students.

The proof of concept system was made so it should be possible to run it directly on the robot base. However, it is worth pointing out that the system does contain some place holder modules. These modules was only made for making the test system have more modules. It was important to test how the controller would act in a situation with more modules then just the navigation module. These placeholder modules are however based on some of the existing modules and could

be replace with the real once when they are ready to be connected to the controller. Some of the existing modules, may have to go through some changes, i.e., they may have to be changed to use actionlib servers. The author has not read through all source code for all other existing modules for the Cyborg, so I don't know how much work this will be, but based on the modules I have looked at, I believe that it should be minimum with changes needed. The future work related to the controller will therefor be to integrate the old modules into the new system.

12. Conclusion

In this thesis a controller module for the NTNU Cyborg has been implemented. The controller module connects to other NTNU Cyborg modules and is controlling them using the ROS actionlib protocol. The controller uses an event driven state machine to ensure that the NTNU Cyborg is in a well defined state and that modules that runs at the same time is not conflicting with each other. Furthermore the controller has a model for the Cyborg's mood, e.g., angry. It uses a PAD emotion state model and has a ROS subscriber for receiving feedback from other modules and a ROS publisher where it publishes the mood to the other modules. The controller is using a simple reward system to decide what it should do when it becomes idle. It selects action that increases the PAD values. This system increases the chance that the Cyborg starts doing things on its own instead of remaining idle.

A navigation module for the NTNU Cyborg was also implemented. The module connects to the NTNU Cyborg as described above, and uses the ROSARNL node for controlling the robot base. This module was selected to be the first module to be integrated with the new controller because it provides features that are relevant for the short term goal of having the Cyborg (robot part) be able to move around in Glassården by the summer of 2017. The module add features such as scheduling the Cyborg to be at a certain location at a given time, allowing a user to ask for direction, allowing the Cyborg to start wander/moving without user interaction (by using ROSARNL).

The controller module, the navigation module and some other minor modules was tested together to see if they worked and to asses if it creates a viable solution. The observation showed that the system worked as expected based on the specification. The proof of concept works good enough to be continued developed and used in the NTNU Cyborg and is providing features that can help reaching the short term goal of the project. ...

Appendix

A. System Requirements and Setup

The test system was running on Ubuntu 16.10. The software for the Pioneer XL must be installed:

- libAria
- baseArnl
- libArnl
- mapper3
- mobileEyes
- mobileSim

The other requirements (dependencies) for the test system can be set up using the script below.

```
#!/bin/bash
echo "Setup script running..."

# Exit if failure
set -e

sudo apt-get update
sudo apt-get upgrade
sudo apt-get install git

# ROS
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc)
    main" > /etc/apt/sources.list.d/ros-latest.list'
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key
    421C365BD9FF1F717815A3895523BAEEB01FA116
sudo apt-get update
sudo apt-get install ros-kinetic-desktop-full
sudo rosdep init
rosdep update
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
echo 'source ~/catkin_ws/devel/setup.bash' >> ~/.bashrc
source ~/.bashrc

# Setup ROS
mkdir -p ~/catkin_ws/src
```

```
cd ~/catkin_ws/src
sudo apt install catkin
catkin_init_workspace
cd ~/catkin_ws/
catkin_make
source ~/catkin_ws/devel/setup.bash
echo $ROS_PACKAGE_PATH

# Install SMACH
sudo apt-get install ros-kinetic-smach

# Other
sudo apt-get install python3-pip
pip3 install npyscreen
pip3 install --upgrade pip
sudo apt-get install python3-pygraphviz
sudo apt-get install python-pygraphviz
sudo apt-get install sqlitebrowser
pip install --upgrade pip

# Speech Output requirements:
sudo apt-get install python-requests
sudo apt install python-pip # This is pip2 for Python 2.7
sudo pip2 install SpeechRecognition
sudo apt-get install sox
pip2 install PyTTSX

# Music Module requirements
sudo apt-get install vlc
pip2 install python-vlc

# Command tool requirements
pip2 install npyscreen

# Put ROSARNL in ~/catkin_ws/src
sudo apt-get install ros-kinetic-move-base
cd ~/catkin_ws/src
git clone https://github.com/MobileRobots/ros-arnl
cd ~/catkin_ws
catkin_make
```

```
64 source ~/catkin_ws/devel/setup.bash
65
66 # Base requirements
67 sudo usermod -a -G dialout $USER
68 # Relogin is required for last cmd to take effect
69 echo "You must logout and back in for userpivilages to take effect..."
70 gnome-session-quit
71 # Alternativly: rebbot
72
73 echo "Setup script ended..."
```

Bibliography

[1] http://wiki.ros.org/actionlib, 11 2016.

[2] http://wiki.ros.org/catkin, 11 2016.

[3] http://wiki.ros.org/catkin/cmakelists.txt, 11 2016.

[4] http://wiki.ros.org/catkin/package.xml, 11 2016.

[5] http://wiki.ros.org/catkin/workspaces, 11 2016.

[6] http://wiki.ros.org/parameter

[7] http://wiki.ros.org/roslaunch, 11 2016.

[8] http://wiki.ros.org/rosservice, 11 2016.

[9] http://wiki.ros.org/rostopic, 10 2016.

[10] http://wiki.ros.org/services, 11 2016.

[11] http://wiki.ros.org/smach, 11 2016.

[12] http://wiki.ros.org/smach/tutorials/user

[13] http://wiki.ros.org/smach_viewer, 11 2016.

[14] http://wiki.ros.org/tools, October 2016.

[15] www.ros.org, 10 2016.

[16] Cyborg follower module - sorce code on github, Januar 2017.

[17] Cyborg selfie module - sorce code on github, 1 2017.

[18] Cyborg simon says module - sorce code on github, Januar 2017.

[19] https://en.wikipedia.org/wiki/pad_emotional_state_model, 1 2017.

[20] https://github.com/mobilerobots/ros-arnl, Februar 2017.

[21] https://sqlite.org/, Februar 2017.

[22] https://sqlite.org/, Februar 2017.

[23] http://wiki.ros.org/, Januar 2017.

[24] http://wiki.ros.org/actionlib/detaileddescription, Januar 2017.

[25] http://wiki.ros.org/ros/installation, Januar 2017.

[26] http://www.kaaj.com/psych/scales/emotion.html#definition, 1 2017.

[27] http://www.ntnu.edu/cyborg, Januar 2017.

[28] TTK4850 EIT-BYGGELANDSBYEN VÅR 2015. Frame for ntnu-cyborg. Technical report, The Norwegian University of Science and Technology, June 2015.

[29] TTK4850 EIT-BYGGELANDSBYEN VÅR 2015. Iris controller card for ntnu-cyborg. Technical report, The Norwegian University of Science and Technology, June 2015.

[30] TTK4850 EIT-BYGGELANDSBYEN VÅR 2015. Ntnu-cyborg mechanical iris. Technical report, The Norwegian University of Science and Technology, June 2015.

[31] Amund Froknestad. Ntnu cyborg: Forbedring av kyborgens evne til dybdesyn. Technical report, Department of Engineering Cybernetics, The Norwegian University of Science and Technology, Desember 2016.

[32] Amund Froknestad. Unpublished master thesis. June 2017.

[33] Martinius Knudsen. The ntnu cyborg: Utvikling og sammensetting av versjon 1.0. project paper, Department of Engineering Cybernetics, desember 2015.

[34] George Konidaris and Andrew Barto. An adaptive robot motivational system. Technical report, Autonomous Learning Laboratory Department of Computer Science University of Massachusetts at Amherst.

[35] Steinar Kraugerud. Ntnu cyborg with communicational abilities. Master's thesis, The Norwegian University of Science and Technology, July 2016.

[36] Kaja Kvello. Implementing a top-level coordinator unit in ros. Technical report, Department of Engineering Cybernetics, Desember 2016.

[37] Stine Lilleborge. Ntnu-cyborg: Oppsøking av sosiale situasjoner. Master's thesis, June 2015.

[38] Jonas Fyrileiv Nævra. The ntnu cyborg 1.0. Master's thesis, Department of Engineering Cybernetics, The Norwegian University of Science and Technology, June 2015.

[39] Håvard Svoen. Integrating graphical face software with a social robot and de- tecting human interest. Master's thesis, The Norwegian University of Science and Technology, June 2016.

[40] Hanne Marie Trelease, Lars-Erik Notevarp Bjørge, Mikkel Sannes Nylend, Per Odlo, Harald Blehr, Odin Oma, and Mats Jønland. Produktrapport - byggelandsbyen (ttk4850) landsbynummer 10. Technical report, Department of Engineering Cybernetics, The Norwegian University of Science and Technology, June 2016.

[41] Jørgen Waløen. The ntnu cyborg: Robot hardware infrastructure. project paper, Department of Engineering Cybernetics, The Norwegian University of Science and Technology, 12 2016.