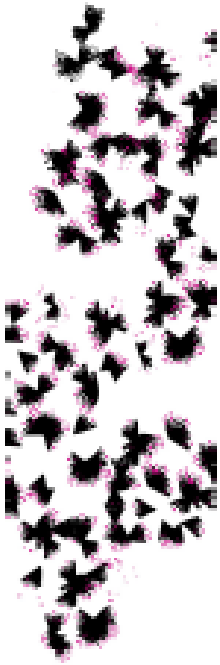

Rail-guided ballast tank inspection robot, Software architecture analysis and enhancement in the ROS environment

P. (Paul) Tieleman

BSc Report

Committee:
Prof.dr.ir. C.H. Slump
D.J. Borgerink, MSc

March 2016

003RAM2016
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

The following report covers the Bachelor's thesis of the study Electrical Engineering. After finishing all the theoretical work of the Bachelor study, this report puts all the learned theory into practice. More specifically, the work carried out took place at the Robotics and Mechatronics (RaM) department at the University of Twente. The topic of the assignment is the RoboShip project, subproject of the SmartBot project, which is a cooperation of different instances with the purpose of automatization of labor intensive jobs and difficult tasks.

At the start of this project, a prototype of the RoboShip project was available. The prototype included a vehicle with a inspecting arm and a variety of sensing equipment. Subject of this thesis is the system architecture on a software level. This includes the control of the robot, which is possible by using a gamepad or a programmable user interface with a couple of buttons and a screen. No explicit description of safe control of the robot was available so at first the prototype was analyzed. After that, an improved version was thought of with respect to user insight in the processes of the robot and safe operation. Implementation of the new system was done next. Finally both the new system and prototype are compared and evaluated.

Table of Contents

1. Introduction

Currently, the maritime sector is still improving regarding automatization. Within this sector, one of the most labor intensive tasks is the inspection of small and dark ballast tanks in ships. Due to the small gaps one has to enter and the lack of fresh air this job is dangerous and very physically demanding. These ballast tanks are used to keep the ship in balance and have to be inspected regularly due to possible leakages and/or oxidation. Salt water is a very aggressive substance on steel, that is why the steel is coated. The task of inspecting personnel is rather easy since they only need to inspect visually and eventually measure the thickness of the coating on the walls. Considering these aspects, the RoboShip project [1] has developed a remote controlled robot which should ease the inspecting job by allowing the personnel to stay in a human friendly room.

The RoboShip project [1] was created by an international consortium of instances, including the University of Twente.

The robot consists of several mechanical and electrical parts. Regarding movability between ballast tanks, the tanks are equipped with a rail system made of composite material. With the aid of controlled wheels on the inspection robot (see Figure 1), the device is able to drive on this rail. A camera is present on the robot, since the user should see the inside of the ballast tanks. The second most crucial part, is the robots arm with several joints holding the coating thickness-sensor. This arm, as well as the rest of the parts of the robot are controlled with a custom made user interface. The user interface includes a screen to display camera images.

1.1. Goal

The subject of this bachelor assignment is the analysis and improvement of the software architecture of the RoboShip project. Until now, the robot has a messy hardware implementation and an unstructured controlling system. This system should be transformed into a robust system. Structure should be present in the design such that errors can easily be solved. Feedback from the robot to the user and insight in the performing activity of the device are part of this assignment.

1.2. Outline

At first, in chapter II an analysis of the present system made. Once previous programming choices become clear, the system is evaluated on the tasks it should be able to fulfill in chapter III. The results from the evaluation are used to set new targets for the software architecture which are implemented later on. Within chapter IV Implementation of the newly designed software has to be evaluated again and compared to the first system in order to clearly see the changes in the system. Finally a conclusion and recommendations to the project are given in respectively chapter V and VI.

2. Analysis of the system

To understand the system on a software environment level, the hardware of the robot is explained in a simple manner. The RoboShip robot can be seen in Figure 1. It can be seen that the Robot consists of various parts. A camera is mounted on an aluminum mount which allows the camera to stow in the case of an obstacle. This is called the pan tilt unit (see Figure 1). Another feature of the pan tilt unit is its ability to rotate in two dimensions. Hereby the camera can see every location above the robot. Due to the lack of light at the location of inspection, the camera has a couple of LED's to illuminate the desired location.

On top of the clamp displayed in Figure 1, a robot arm is mounted. Ideally the arm is controlled by means of pointing to a position displayed on the interface by the user. A thickness-sensor is mounted on the tip of the arm. This sensor is used to measure the thickness of the coating at a given location.

The earlier mentioned arm is rather large so some sort of stabilizing mechanism has to be used in order to retain the robot on the same position on the rail when the arm is operational. The mechanism used in this project is a mechanical clamp which can be operated automatically. The clamp is displayed on the bottom of Figure 1.

The control unit of the robot contains several electronic components such as drivers, the CPU and communication converters. This unit is covered under a plastic cover in the middle of Figure 1.

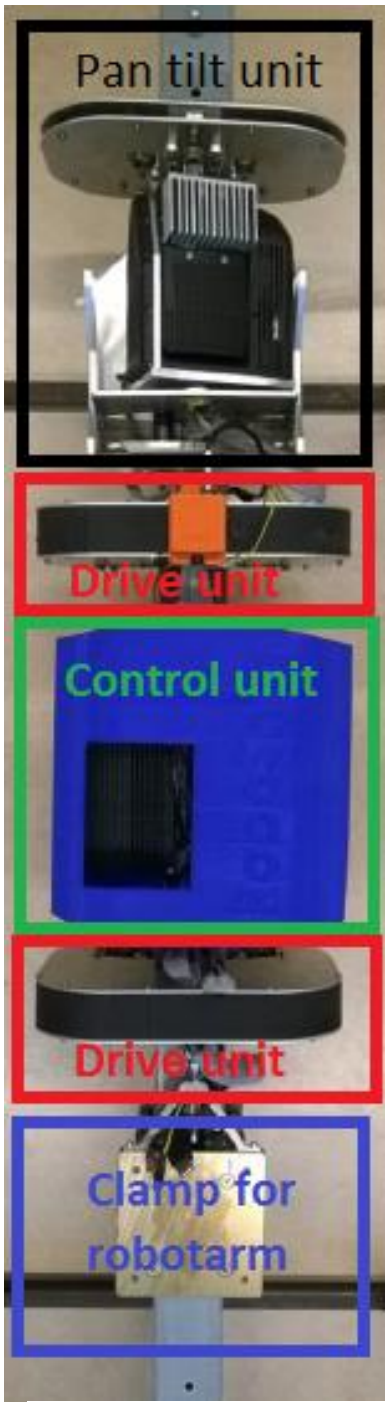

Figure 1: RoboShip hardware

2.1. General software architecture

The software environment used for the robot is ROS. ROS is an open source tool which provides libraries to help design robot applications. This saves a lot of programming and allows different coding language compatibility.

2.1.1. Modules in the system

The total system in ROS consists of a number of nodes, while topics are a way of communication between these nodes. This communication relies on the publisher/subscriber principle [3]. In the given system each module has a separate node. The main loop of execution is located in the “highlevel” package. This architecture has some great advantages, such as a good overview of the different modules. Overview of the modules is created by the separation of the exact protocol of the functions from the controlling system. These modules are parts of the robot, for example the camera. The overview is also created by the close relationship between hardware and software, for example the faulhaber driver used to move the robot on the rail has its own node.

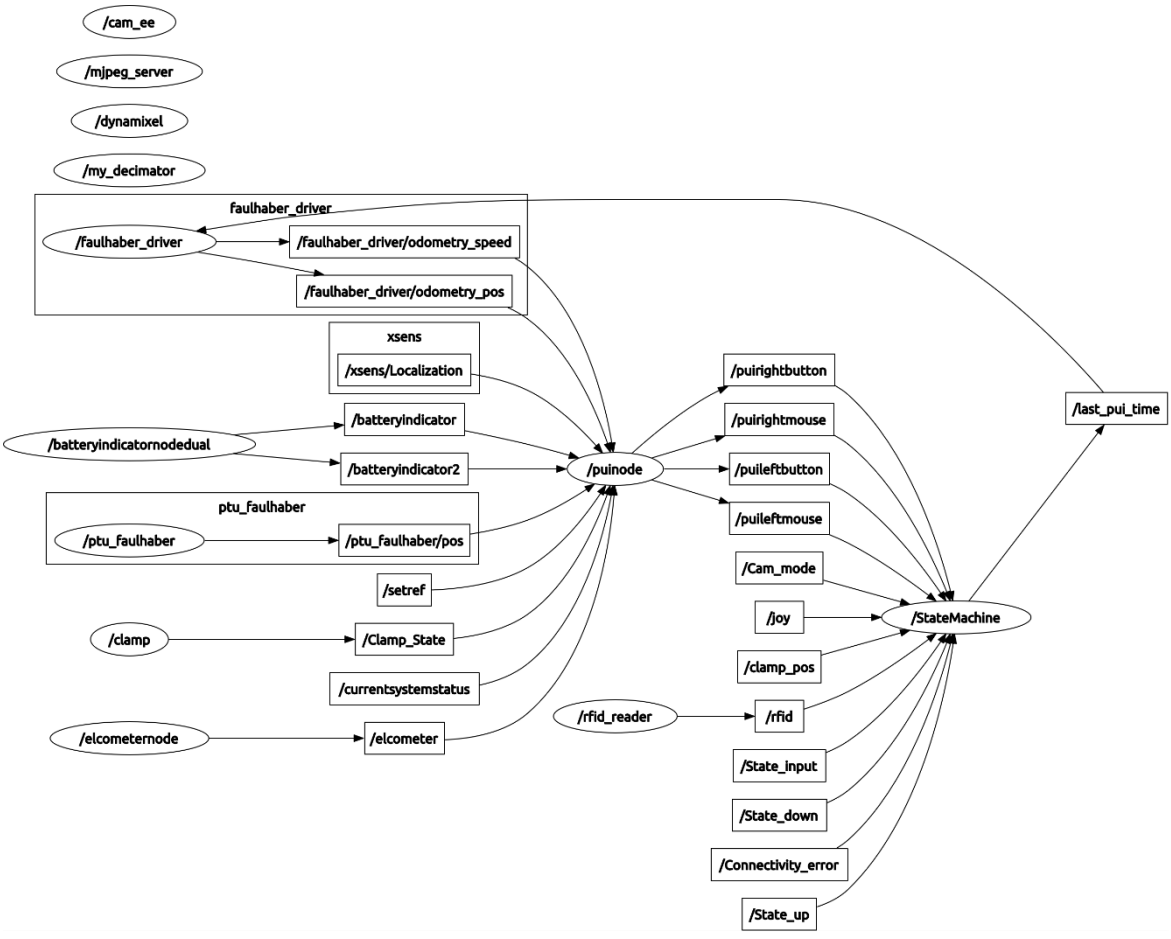

Figure 2: ROS structure of the given system

The structure of the system on the computational level of abstraction is displayed in Figure 2. This plot is directly obtained from the rqt command in ROS. The unchanged system consists of several nodes, displayed in oval shapes, and connecting topics between these nodes, displayed in rectangles. The

rectangles containing more than one object have subtopics. These can be used to exchange more than two types of messages between nodes. It should be noticed that there are some unconnected nodes and that all the information is passed through the “puinode”.

StateMachine

The StateMachine node is the controlling node of the system. Its description is situated in the highlevel package. Various subscriptions were created to this node but are yet unused. Nevertheless these subscriptions can be used in this project to improve the controlling system. One of the usable subscriptions is for example the connectivity_error which can be used to detect connectivity errors.

Rfid_reader

Since spatial positioning within the ballast tank, rfid reading is meant to fulfill that purpose. The node needed is not present in the rqt graph in Figure 2 but should of course be implemented.

Puinode

This is the remote PUI (Programmable User Interface). The PUI consists of a combination of joysticks, buttons and a screen.

Elcometernode

The robot has a inductance sensor for measuring the coating thickness. The process of this measurement is located within this node.

Clamp

The robot has a rail clamp to maintain the arm stable when inspecting. Clamp is the corresponding node for releasing or engaging the clamp.

Ptu_faulhaber

While using the robot, a camera is used to provide vision for the operator. The camera can either be in stow or in panoramic mode. Those two modes are necessary due to the crash possibility of the ballast tank structure and the camera itself. Changing the mode of the camera is accomplished by communicating with the ptu_faulhaber node.

Batteryindicatornodedual

The robot has two batteries to allow wireless operation on the rail. The status of those batteries can be read from this node.

Faulhaber_driver

The actual movement of the robot on the rail is processed in the faulhaber_driver node. The actual speed can be read from this node which allows movement along the rail.

My_decimator

This is an unused down sampling library intended to be used to process camera data.

Dynamixel

Dynamixel is an unconnected and unused node which is not mentioned in the highlevel package. Its intention is to control the manipulator arm.

Mjpeg_server

Unused node which should be used to provide the PUI camera images.

Cam_ee

Camera images at the end of the manipulator arm are provided by means of this node.

Joy_node

This is the node from the Logitech controller. The Logitech controller is controlled by a standard protocol [4].

2.2. Main program

In the source file of the "highlevel" package a C++ executable was created which then creates a node StateMachine and controls the rest of the system. This is not the case as can be seen in Figure 2. In fact, all the nodes communicate directly with the Puinode.

The program has been hacked together resulting in code which is very hard to understand. A few errors are present such as the ability to move the camera while it is in stow mode. These errors probably occurred due to the lack of software structure. An important structural problem is the "joy_callback" function which contains the whole control of the vehicle. An inside look into the working principle of the "joy_callback" function is given in Figure 3.

This function uses the input from the gamepad or programmable user interface to directly control the robot. Meaning the state of the system does not matter when controlling the robot. When the joyCallback function is started the driving speed is always set to the motion of the left joystick. After that, the camera is controlled by the right joystick. The process just described is used for every function in the system. See Figure 3 for a graphical view of this principle of operation.

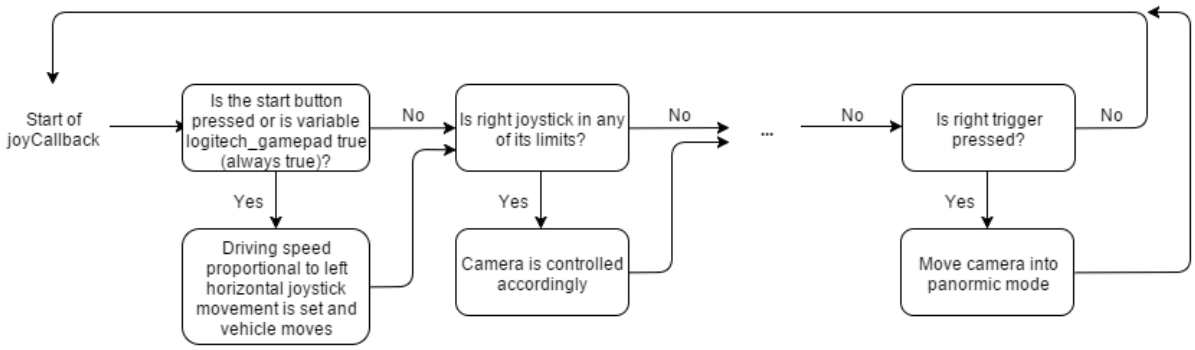

Figure 3: Working principle of "joy_callback"

2.3. Testing the original system

The system is tested for proper functioning by connecting a gamepad as user interface. The robots operation is tested on a single two meter length of rail. While testing, the robots response is observed when using all the possible buttons or joysticks of the gamepad.

In addition to the testing on a single piece of rail, the robot was also tested in a container with an incorporated rail system.

When driving the vehicle, a delay of about one to seven seconds is encountered when changing the vehicle speed. In addition, driving is not possible after not changing the value of the joystick for ten seconds. The clamp is engaged whenever the camera is in stow mode and the robot does not respond when certain buttons are pressed. Aside from these dubious features of the vehicle, there is the camera control which is operational in the stow position of the camera. This could result in a collision between the camera and its stowing mechanism, since there is not enough space for the camera to rotate in the stow mode.

Beside these hardware problems there is the difficulty of controlling the vehicle due to the sudden stops of the vehicle and crashing of the software.

3. Design

3.1. Requirements of the new system

As mentioned in chapter 2, there are some errors in the software of the RoboShip robot that need to be fixed. Summarizing these errors we get the following list:

- Lack of software structure as well as very long coding bulks
- Camera control while stowing
- Delay of driving control on the rail
- Sudden crashes of the program

Besides these errors the following improvements are suggested:

- Introspection from the user into the software of the robot. This means that the user should be able to see what the robot is doing on a software level.
- Smoother camera control, since until now the camera is simply controlled by turning the servos, of the camera mount, on or off. It would be easier to control the camera position proportional to the amount of movement of the joystick.
- Automatic stowing of the camera in case of obstacles.
- Separating the driving and inspecting functionality by means of a state machine. A state machine allows a better overview for user and programmer.
- Battery level measurement, automatic shut down and save procedure when a certain voltage is reached.
- Debug function whenever an error needs to be solved. Accessing this function happens by using a debug trigger.
- Connectivity check at start up: When starting up the system the connections to the modules of the system are checked such that the robot knows it is ready to be used.

3.2. Software Architecture

In order to improve the old system, a general perspective is needed. When looking at the different general modes of operation, the driving and inspecting modes come to mind. Camera control should be possible within both modes such that the user is able to either see the path of movement or see the coating on the steel when inspecting. The clamp on the RoboShip is not to be controlled manually since this is primarily automated with ease and secondly because attempting to drive while manually operating the clamp could result in damage. In fact, the clamp should never be operated when the robot is moving. Concluding this general description, two main states are identified; driving and inspecting. The implementation of states makes the ongoing processes in the robot easier to understand. This due to the fact that, for example in the driving state there should not be any movement of the arm. After instruction of the personnel this is an easy fact to remember. In comparison to the use of a state machine there is the implementation of if-statements. A disadvantage of if-statements is that they give no insight from the user in the system since there is nothing to be displayed on the PUI. In a system with states therein a simple state machine can be displayed with an indication of the current state.

Right from the start of this thesis, the vision of a state machine seemed to be the most adequate solution.

In addition to the “driving” and “inspecting” states an “idle” state, “start up” state and a “shutdown” state are created. The “idle” state is used as a transition between the other states that the robot has a “stand by” function. In the “stand by” function the system is just waiting for a command of the operator. This saves energy and adds some safety due to the better separation between driving and inspecting. In case of inclusion of the idle state, the state machine has to transition three times. At start up, the connections should be checked so that every module of the RoboShip is known to be accessible. In order to perform this check, a startup state seems to be the perfect solution. Then, after a successful check, the system switches to the idle state. In order to shut down the RoboShip, the operating system should end all active processes.

A simple representation of the system is displayed in Figure 4.

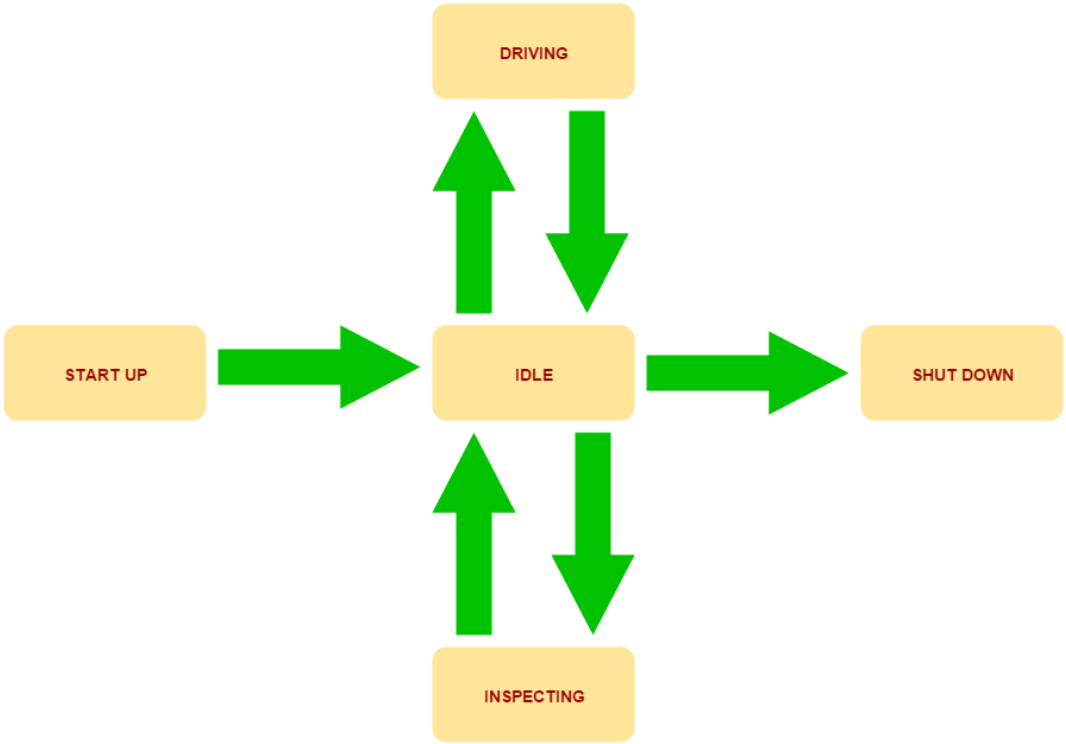

Figure 4: Simple system description

The new state machine should be robust and handle errors during its operation. Therefor the system previously described is designed with the aid of UML (UnifiedModellingLanguage). Graphically the designed system looks as in Appendix B.

For a simple insight in the UML state machine, see Figure 5.

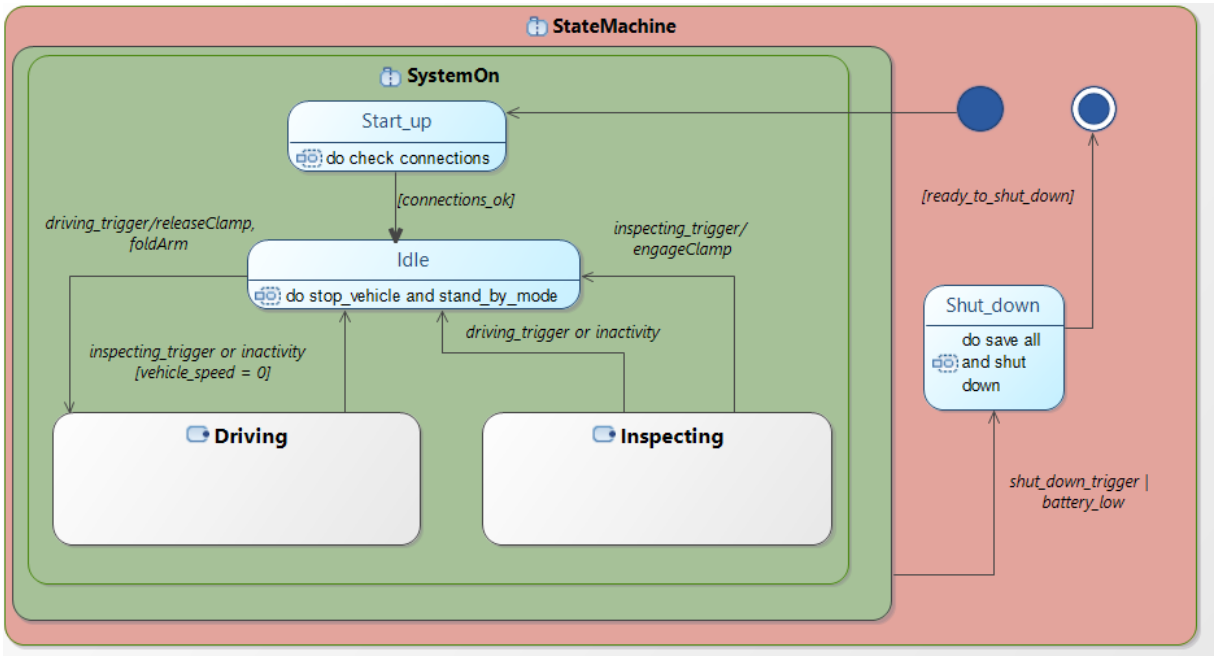

Figure 5: State machine in general view

3.2.1. States

Finalizing these thoughts a more elaborated description of the states is given.

Start_up

The start_up state is used to initialize the system. In order to initiate the system properly, the system needs to be checked prior to enabling its functions. At first all the connections to the different modules of the system such as the camera are tested. After sending a message to the nodes, the response or its absence determines if there is a connection. If not all modules are connected, the system will automatically turn into debugging mode.

Idle

The idle state is used as a transitional state between the driving and inspecting state. After finishing the ending processes of the driving state and inspecting state the robot is set to the idle state. From here driving and inspecting can be reached. The idle state is mainly used as a waiting state.

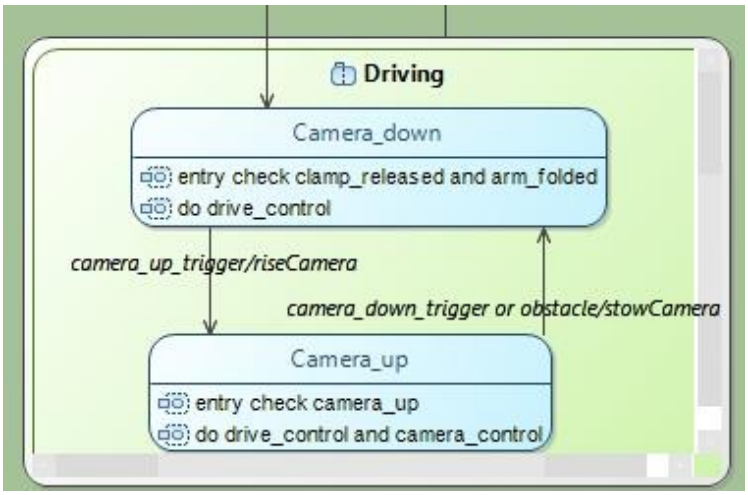

Figure 6: Driving UML state

Driving

The driving state consists of two sub states (see Figure 6). The reason for allowing the usage of the camera is obvious: the inspecting employee should see where the RoboShip is heading to. Stowing the camera is necessary since the gaps in the walls of the ballast tanks in the test environment located at the University of Twente. These gaps are the transitions between compartments in the ballast tank. Stowing the camera at these transitions could be done automatically when the robot knows where it is located in the ship. The position is determined by the project of Steven Gies.The lack of time didn't permit the design and testing of this rather useful option.

Within the driving state there are some conditions that should always be fulfilled. These conditions are the "clamp_released" and the "arm_folded". Both conditions are observed with the aid of the "Clamp_State" and the "Arm_State" topic. Arm_state is the topic that is being designed by Laurie Overbeek, who is responsible for the arm control.

Camera_up

When the camera trigger is applied, the camera is set to panoramic mode by calling the service that controls the mechanism of lifting the camera. After doing so, the topic "ptu_faulhaber/…" is read to see if the camera really is in the panoramic mode. If lifting the camera fails, the debug mode is initiated. If lifting the camera succeeds, the camera can be controlled with the right

joystick of the gamepad. The control of the camera is done via the “dynamixel/setPos” service. Camera images aren’t read out in the script yet, but can be easily added by subscribing to the “cam_ee” topic. The implementation of the video feature is not considered as essential in this bachelor thesis since the system design is tested with a gamepad and not the actual remote control interface.

Camera_down

Stowing the camera is accomplished by settting the position of the ptu mechanism via the “ptu_faulhaber/..” service. In order to know if this action succeeded, the topic “Camera_state” read out. Just as in the “Camera_up” state, the debug mode is initiated whenever the stowing action fails.

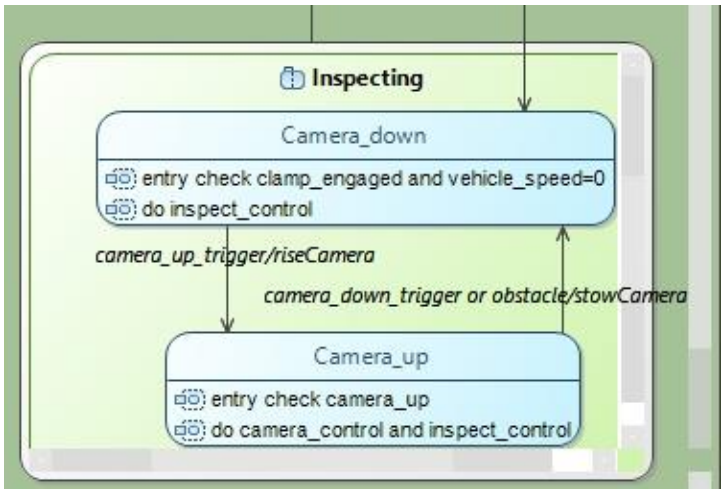

Figure 7: Inspecting UML state

Inspecting

Within this state the clamp should be engaged and obviously the vehicle should be at rest. The vehicle speed is determined via the faulhaber/odometry_speed service which calculates an average speed of the 4 wheels. The camera is an important feature within the Inspecting state since the operating personnel should be able to see the inside of the ballast tanks. The inspecting state in UML code looks as in Figure 7.

Camera_up

This substate works exactly the same as the substate Camera_up in the Driving state.

Camera_down

This substate works exactly the same as the substate Camera_down in the Driving state.

Shut_down

Whenever the “shut_down” trigger is applied or the battery has less than 10%, the state of the system is set to “Shut_down”. Battery level of both batteries is measured via two circuit boards. The script uses 29V as 100% and 22,5V as 0%. The “Shut_down” state is necessary to end the control of the RoboShip in a neat way.

3.2.2. Debug

Debug is necessary to control the robot in situations that the user needs to bypass the controlling system. This means that the user is able to control every function of the robot without having to worry about conditions that should be met. The debugging mode could, for example, be very useful when the arm has to be folded in a different manner before driving.

Since debug should be usable in all states, it has to be a mode and is thereby not present in the state machine.

4. Implementation

While implementing the changes proposed, the system is divided into three scripts located in the source folder of the new ROS package. The program consists of one main script used to implement a SMACH structure and one script to read from topics and one script to call services. All three scripts are coded in Python language since this is the language used for SMACH. SMACH [5] is a package used for the design of state machines in ROS. It is the only maintained package at the time of this project on the official ROS website [1]. This fact and the advantage of easy introspection into the state machine via the smach_viewer package are the reason for choosing SMACH.

4.1. User interface

After launching the smach_viewer package, the state machine is automatically created graphically. In the introspection interface the current state is visible by means of the use of green filling color and a thicker enclosing line (see Figure 8). Changing the graphical view is not possible in SMACH.

Within the user interface there are a couple of buttons and couple of things to be noted. As can be seen in the figure, the interface mentions “Path not available”. The reason for this text is the choice of the path in the upper left corner of the window, which is not chosen yet. Different paths can be chosen to look inside the state machine with more detail (the inspect state in Figure 9 and the drive state in Figure 10).

The choice of the depth is closely related to the path choice since it determines on which level the user is observing. Label width and the “Show Implicit” button are unused functions of the interface.

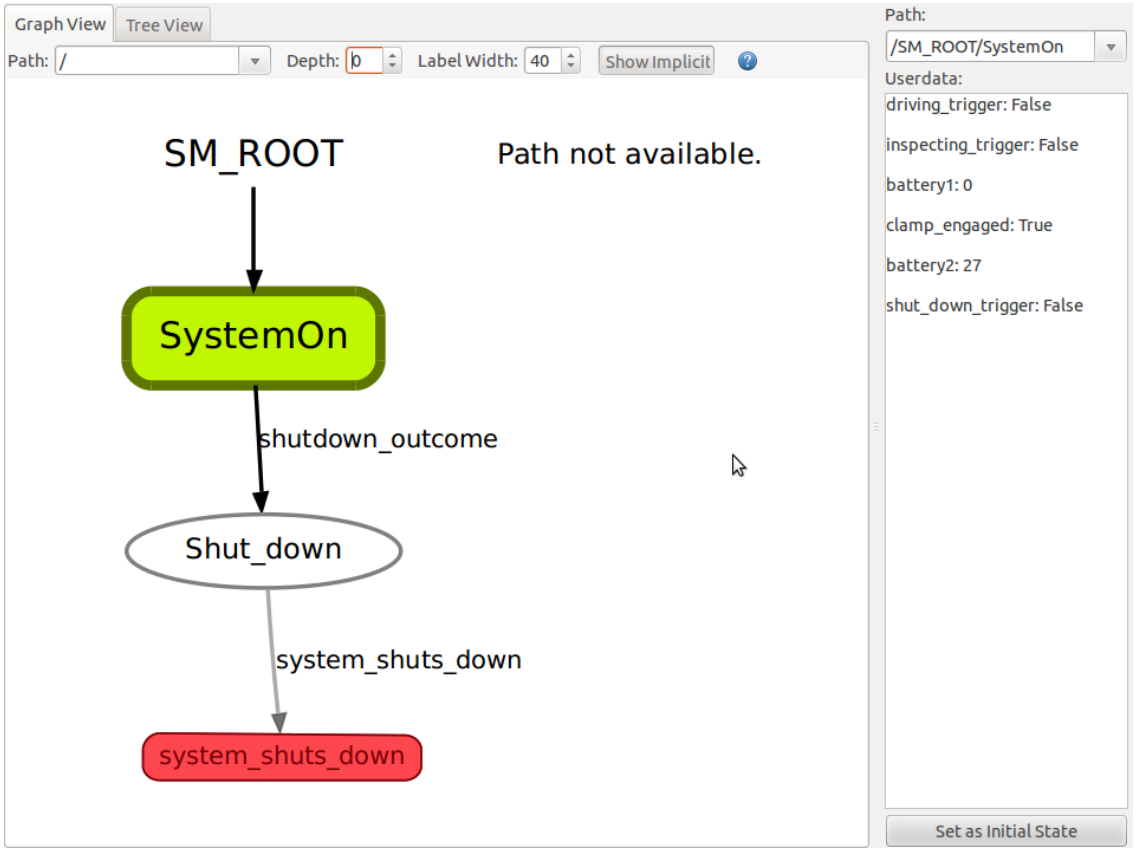

Figure 8: User interface

The outcomes listed in the introspection windows are simply the names of the transitions between the states. They could be chosen differently by for example mentioning the corresponding trigger to make the introspection more understandable to the user.

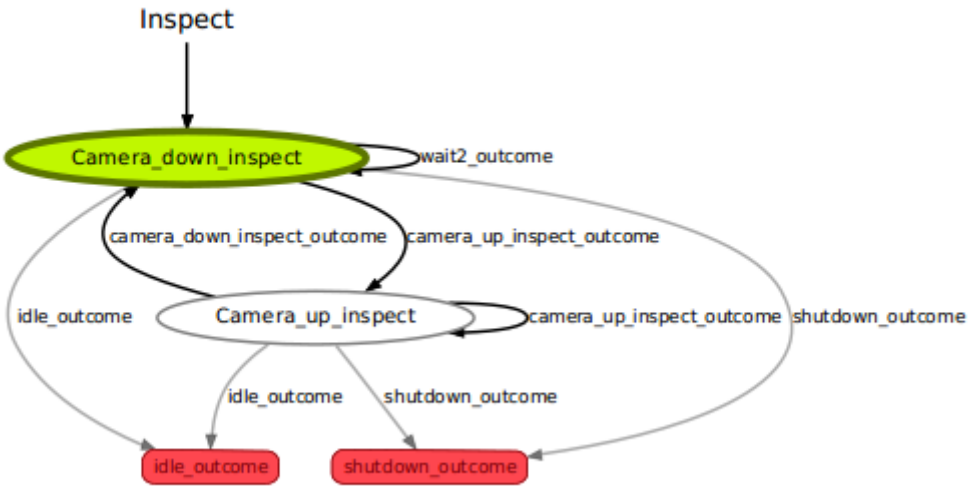

Figure 9: Inspect state

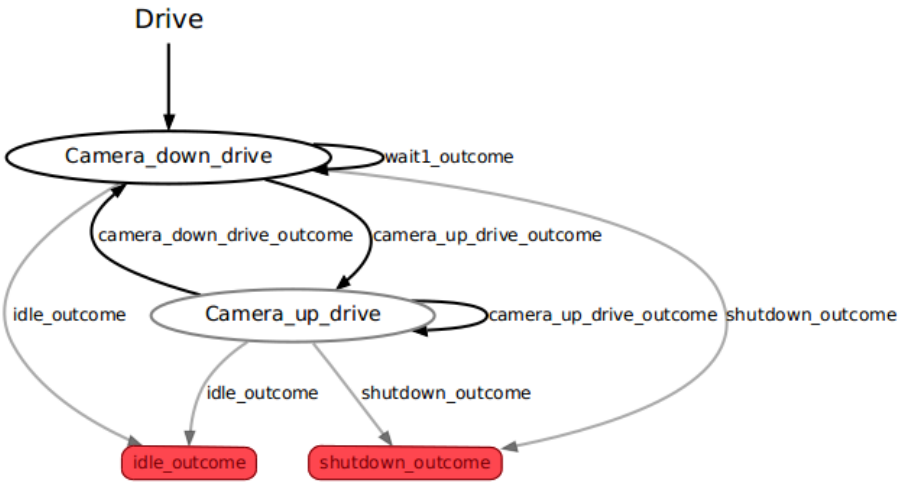

Figure 10: Drive state

An equally useful feature of the smach_viewer interface is the display of userdata (see Figure 11). Userdata is real-time data from the system. The following data sources are chosen to be userdata, since they could be useful for debugging and operation of the robot:

1) clamp_engaged: Boolean which is true if the clamp is engaged and false when the clamp is disengaged.

2) vehicle_speed_cm_per_s: Integer equal to the speed of the vehicle in meters per second.

3) driving_trigger, inspecting_trigger, shut_down_trigger: Booleans that indicate the state of the respective trigger.

4) battery1, battery2: Integers which display the voltage of the battery in percentage to its voltage range from 17.5V to 29.5V.

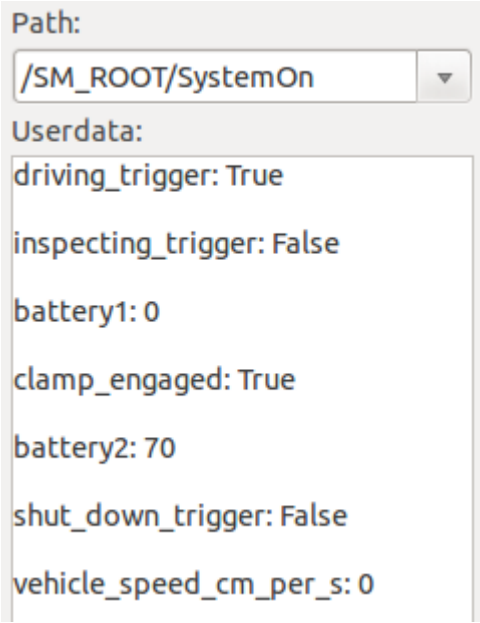

Figure 11: Userdata in SystemOn state machine

4.2. Program

The created program consists of three scripts forming the new software of the RoboShip. For a thorough explanation of the joystick controls see Appendix A.3.

4.2.1. State_Machine.Py

This is the main script containing the structure of the state machine. At first, all the sources are enlisted. Packages such as rospy, smach, smach_ros and time are imported. The reason for importing both smach and smach_ros, lies in the fact that the last package contains the necessities to use smach viewer. Rospy is the client library in python for ROS. After importing these packages, the service_calls and subscriber_functions scripts are imported such that the functions described therein can be used in the state machine. Aside from these scripts, the Joy script is imported to process data from the gamepad too.

Next, the different states are defined in class types. A class can contain functions and is very useful in this program since it allows some structure. The class is first initiated in "def __init__(self)". Then, the class is defined to be a smach state by using the following code:

```
smach.State.__init__(self, outcomes=['…'], input_keys=['…'], output_keys=['…'])
```

In this code line the state is defined to have some outcomes which are used as state transitions and some input and output keys which are respectively the input and output data of the state. The transition names can later be used as quick reminders on what is happening. Within the initiation of the state, objects of the state can be defined. An example is the "shut_down_trigger" which is initiated to be false, since otherwise the system would shut down immediately. This is accomplished with the "self." argument:

```
self.shut_down_trigger = False
```

Then, the class is subscribed to a certain topic as such as the joy topic:

```
self.subscriber = rospy.Subscriber('/joy', Joy, self.callback)
```

By doing this directly in the definition of the state the program is checking the joy topic on each iteration. This is necessary for the gamepad, since the controller should be able to control the robot on each iteration. The rospy.Subscriber function here subscribes to the topic "joy" of message type "Joy" and inserts the message in the callback function. The callback function in question is "joy_callback" and is located in the subscriber_functions script.

After the initiation of the state, "*def execute(self, userdata)*" determines the tasks to be executed which could use some userdata. In the state_machine script, the userdata is set to be equal to the objects created in each state such that the userdata can be used in the smach viewer. Passing userdata between states did not function. All variables that have to be passed between states are defined as objects.

The rospy package offers some debug info which could later be used to implement in the debug mode. That is why "*rospy.loginfo('…')*" is already implemented in the code. There are other types of feedback from the system with other severity such as warning or error type of messages. These messages should be visible used whenever the debug mode is active, then the user knows exactly where the program is experiencing problems.

Switching between states is accomplished with the aid of if statements. These statements seem to be quite unstructured, that is why case arguments could be used. Even though this is an option, if statements are chosen because often there is a need for hierarchy for conditions. Hierarchy in case statements is a possibility too, but it does not seem to be the simplest way. An example of the need for hierarchy for the conditions, is the battery state, which determines if the state is going to proceed at all.

The main execution of the program is located in the main function. Within this main function a unique node is intiated:

```
rospy.init_node('StateMachine', anonymous=True)
```

This node is now the controlling node of the RoboShip. After this node creation a SMACH state machine called sm_top is created. The state machine has “output_keys” equal to the userdata. These variables can be seen in the SMACH viewer. The sm_top container contains substate machine sm_subtop containing the “Start_up”, “Idle”, “Drive” and “Inspect” state. The “Drive” and “Inspect” state contain another substate machine both containing “camera_up” and “camera_down”.

Each state is added via the “smach.StateMachine.add” command:

```
smach.StateMachine.add('Idle', Idle(), transitions={'driving_outcome':'Drive'})
```

The meaning of the previous code is that the state idle is added to the state machine with function Idle() and possible transition to the Drive state if the state returns “driving_outcome”.

In order to create an introspection server and execute the SMACH state machine, the script contains the following lines of code:

```
sis = smach_ros.IntrospectionServer('StateMachine_server', sm_top, '/SM_ROOT')
      sis.start()
      outcome = sm_top.execute()
```

This initializes the server with ROS introspection namespace “StateMachine_server”. The state machine displayed in the smach viewer is sm_top and the name is SM_ROOT.

At last, the main loop is executed until the application is stopped via the ctrl+C keys.

4.2.2. Subscriber_functions.py

Within the subscriber_functions.py script the connections to the topics are established. This script contains functions which are called from the main script, State_Machine.py. Suppose that for example the clamp state needs to be known, then the function clamp_state() inside this script is called. The function clamp_state() returns true or false whether the clamp is respectively engaged or disengaged. The function in case looks as following:

```
def clamp_state():
	global clampEngaged
	def callback(data):
		global clampEngaged
		clampEngaged = data.data
		rospy.loginfo(rospy.get_caller_id() + "I heard clamp_engaged = %s",
data.data)
		time.sleep(0.1)
		return clampEngaged
	# subscribe to topic clamp state
	rospy.Subscriber("/Clamp_State", Bool, callback)
	return clampEngaged
```

After opening the function the global “clampEngaged” is created such that its value can be returned later on. It was intended to do this in a cleaner way since globals can become quite tricky at some point (they override the system). The globals can not be replaced by using objects of the function because they can not be read from the main script.A time delay of 0,1 seconds is implemented because the clamp state does not need to be read out that often. The “rospy.Subscriber(“/Clamp_State”, Bool, callback)” function subscribes to the topic “Clamp_State” with message type bool and returns the messages to the callback function.

The “subscriber_functions” script is used to read the buttons of the gamepad. In order to use all the buttons, separate bool type variables are created for each button. The movement of the joysticks of the gamepad returns an integer value between 0 and 1. Both joysticks can be moved either horizontally or vertically. This means that there are 4 integer values for the axes assigned to an array.

4.2.3. Service_calls.py

In this script service calls are executed. Just as in the “subscriber_functions” script, functions are created in order to communicate with services. These functions are:

controlCamera

Within this function the camera can be controlled by using the joystick on the gamepad. First the program has to wait for the service “setPosition” and create a proxy for both the “setPosition“ and “stop” services of types “setPosition” and “stop” type respectively. The name of the service and the type of message are chosen to be the same so that programming is easier, and there can not be any confusion with other types. Whenever the service is available, the right stick of the gamepad is checked for activity. Then if one of the 4 possible joystick variables in the array is set to 1, meaning joystick completely pulled or pushed, the position of the camera is set to the limit of the corresponding motion direction. If there is no activity the stop service is called, so that the camera stops its motion. The process of moving the camera could be improved since using the servos only at maximum speed is not easy to handle. Improving this process is not possible since the servos only contain the option to be controlled by means of position. It may have been possible to change the script of the servos but no further attention was put into this minor detail due to the usage of C++ coding language in this script. Learning the C++ language would have taken too much time. If the service communication were to fail, an information message is displayed in the terminal with the service in case.

riseCamera

Basically the process of calling the service is the same for each function within this script. In order to rise the camera from its stow position, the service “panoramicPos” of namespace “ptu_faulhaber” is used. An empty message is used to call this service. After this call, the camera script moves the camera into stow positon.

engageClamp

This function is used to engage the clamp. First, the program waits for the service “ClampSendCommand” to be published. Then, a proxy is created to establish the connection. If the service is published, the clamp servo is set to position equal to 300 (engaged). Then the program is set to sleep for 8 seconds which is approximately the time needed to engage the clamp. If the service communication does not succeed, the failed service is displayed in the terminal in the form of a ROS-info message.

releaseClamp

The function “releaseClamp” consists of nearly the same code as the “engageClamp” function, except for the different position in the “SetPos” command.

4.3. Evaluation

The new software on the RoboShip works and does have the wanted introspection. Moving the camera in stow mode is not possible anymore. Nevertheless, the system does not yet control the arm, since the arm itself is not yet finished. Controlling the inspecting arm is part of the project of Laurie Overbeek.

While implementing the new software design, the “GetBatteryStatus” topic sometimes returns values of 100 instead of the expected 0. This is possibly caused by the gate capacity of the measuring circuitry. The appearance of this error is furthermore the reason for not including the battery_low userdata or automatic shutdown at low battery level in the program.

It is further observed that while moving the RoboShip, a delay between gamepad operation and vehicle motion occurs. By mindfully using rosloginfo commands, this delay can be measured in the rosconsole. An average delay of 0,3 seconds is found in the calling of the faulhaber_driver service. Even though, the observed delay is in the range of approximately one or two seconds, so probably the communication to the drivers is very slow.

During the test a peculiar error in the new software is noticed. The RoboShip does not stop driving when the camera is put to stow mode or changed to panoramic mode until the camera has reached the desired mode. This is caused by the fact that the faulhaber drivers are set to a certain speed and simultaneously the ptu_faulhaber service is called. The ptu_faulhaber service stops when the camera has reached its desired limit and while this process is going on, the speed set for the RoboShip cannot be changed. This because when calling a service, the server node will handle one call at a time and queue the rest. Of course this can be solved easily by first setting the speed called to zero before changing the cameras mode.

Due to the lack of time during this assignment the debug mode as well as the connection check have not been implemented.

5. Conclusion & Recommendations

The RoboShip, with its new software architecture, is found to be an improvement compared to the old system. There is still debug and startup check missing within the software but the addition of introspection is very useful. Introspection is found to be of great help due to the direct insight into the parameters of the system. The structure of the software is easier to understand and seems to have more structure. Structure has clearly improved comparing the node plot in Figure 2 with the state machine in Appendix A.1.

Taking all these facts into consideration, the new software is not yet perfect but a lot more robust than before.

Controlling the robot arm is still done with a joystick, but setting the position of the arm could additionally be done by using a laser pointer which indicates the destination of the inspecting arm.

The “Shut_down” state could log all the variables and also the state of the robot before the program is ended. Then, when the system is initiated, this information is still available. Usable information can be the reason of shutdown for example battery level was too low or “shut_down” trigger was applied.

References

[1] RoboShip website, http://www.smartbot.eu/nl/roboship/

[2] ROS wiki website, http://wiki.ros.org/

[3] ROS nodes, http://wiki.ros.org/Nodes

[4] Standard gamepad control ROS package, http://wiki.ros.org/joy/

[5] State machine construction ROS package, http://wiki.ros.org/smach

Appendix

A.1. UML state machine

The system design is performed in the UML language. Shown is a state machine with a blue circle as program start and a blue/white circle as end of the program. There is one substate machine “SystemON” and two subsubstate machines “Driving” and “Inspecting” inside the system.

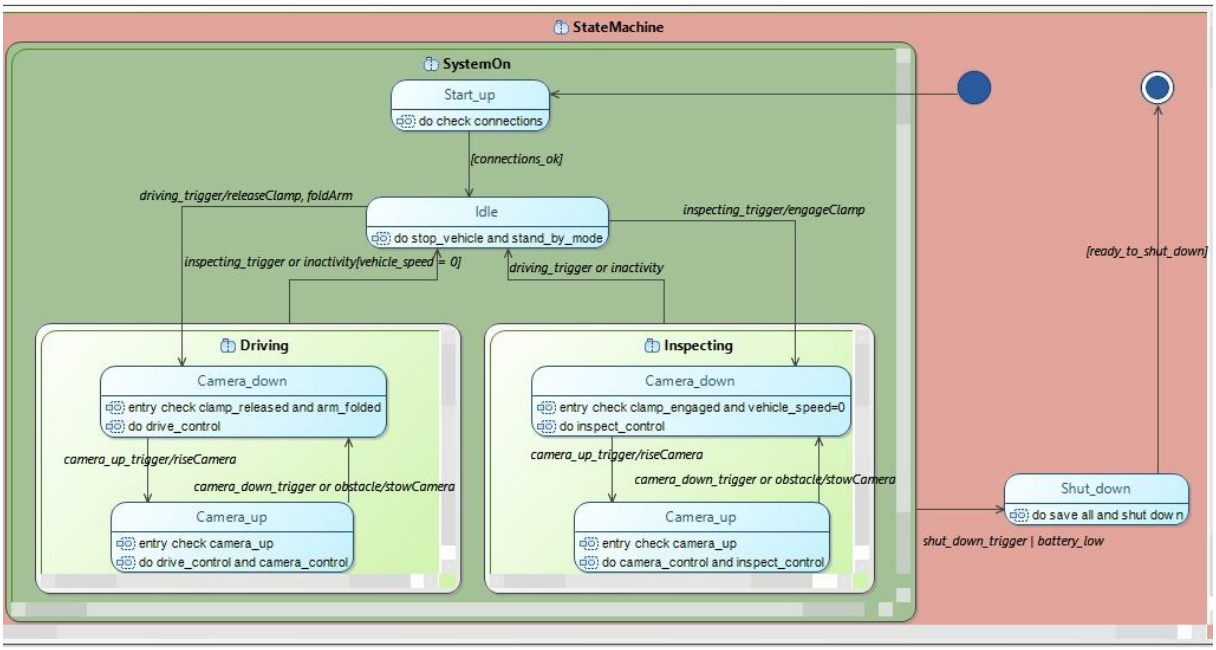

Figure 12: UML design of the state machine

A.2. ROS commands

When loading the newly created software a couple of questions could arise. That is why this FAQ is added to the bachelor assignment

How to launch the introspection interface?

The smach_viewer package finds an active introspection server and displays its data in the interface.

```
$ rosrun smach_viewer smach_viewer.py
```

How to run the new statemachine?

```
$ rosrun paul_highlevel state_machine.py
```

How to run the joy package which is used to communicate with the gamepad?

```
$ rosrun joy joy_node
```

If the gate of the communication is incorrect, it can be found by using the following command:

```
$ sudo jstest /dev/input/js1
```

The gate could of course be js0 or js2 as well, etc. If the gamepad is used, the terminal shows the change in the message received. When the correct input gate is found the default gate for the gamepad can be set, for example js1:

```
$ rosparam set joy_node/dev "/dev/input/js1"
```

How to get the smach_viewer package?

```
$ sudo apt-get install ros-hydro-smach-viewer
```

How to access internet on the system?

```
$ sudo /etc/init.d/networking stop
$ ps aux | grep Net
```

This displays two lines of code with two roots. Those roots should be killed (for example 6220 and 7078):

```
$ sudo kill -9 6220 7078
$ sudo /etc/init.d/networking start
```

A.3. Gamepad commands

In order to control the RoboShip, the Logitech gamepad is used. With the aid of the following buttons and joysticks, the software is able to either change state or to perform an action. The Logitech gamepad has a button to change the addresses of the output array. In this Project the gamepad has to be set to mode D.

Button	Function	Condition
A	Switch to driving mode	Battery level is ok
B	Shut the system down	System is ready to shut down
Y	Switch to inspecting mode	Battery level is ok, vehicle speed is 0
Right trigger	Move the camera into panoramic mode	Battery level is ok
Left trigger	Move the camera into stow mode	Battery level is ok

Joystick	Function	Condition
Right horizontal	Turning the camera on the horizontal plane	Battery level is ok, camera is in panoramic mode
Right vertical	Turning the camera on the vertical plane	Battery level is ok, camera is in panoramic mode
Left horizontal	Driving of the robot	Battery level is ok