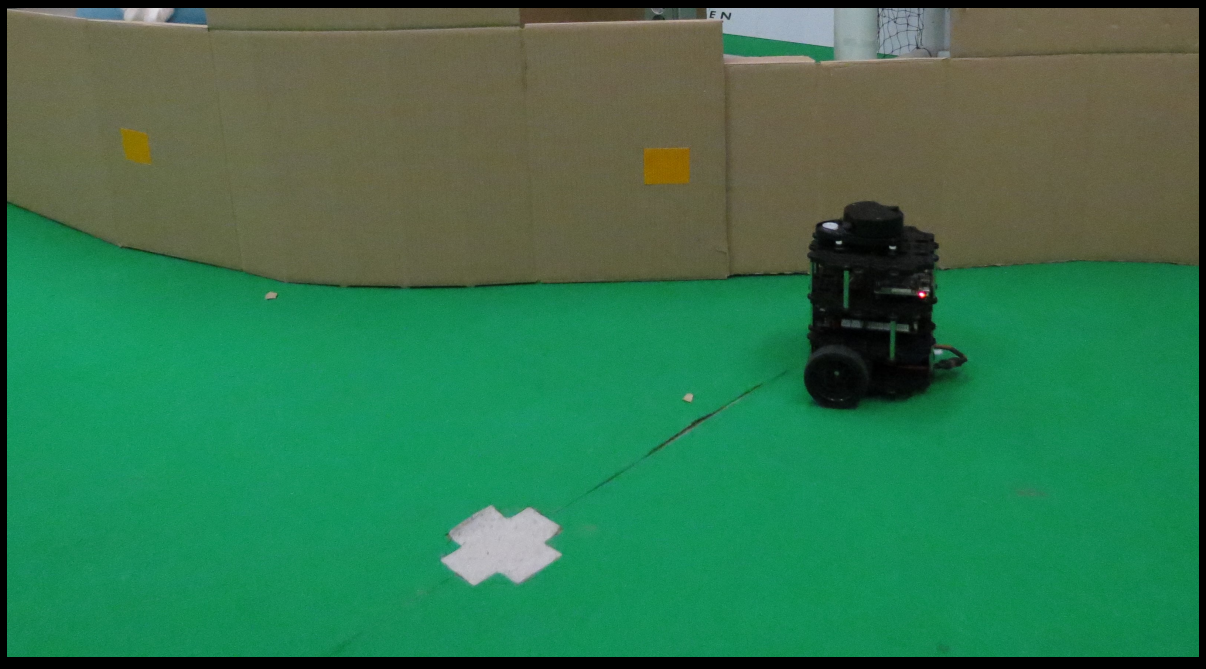
Application

Programming for Embedded Systems in Education on TurtleBot3 using Statecharts



Mirka Schoute

Layout: typeset by the author using LaTeX.
Cover illustration: Photo by Mirka Schoute

Application Programming for Embedded Systems in Education on TurtleBot3 using Statecharts

Mirka Schoute
11868643

Bachelor thesis
Credits: 18 EC

Bachelor *Kunstmatige Intelligentie*

University of Amsterdam
Faculty of Science
Science Park 904
1098 XH Amsterdam

Supervisors
Prof. dr. K.B. Akesson and ing E.H. Steffens

Informatics Institute
Faculty of Science
University of Amsterdam
Science Park 904
1098 XH Amsterdam

June 26th, 2020

ABSTRACT

This thesis examines the possibility of using the TurtleBot3 Burger and YAKINDU Statechart tools to teach application programming for embedded systems. This is done by comparing these new tools to the Lego Mindstorm EV3 and Mathworks Stateflow, which were previously used to teach in this context. Development to test the new tools is described. In the development, a ROS 2 interface that can be connected to the interface of the generated code from YAKINDU is described. Over development the level of abstraction that is suitable for using the TurtleBot in education in this context is determined. This together will clarify the possibility of use of the Burger and YAKINDU Statechart tools for education in application programming for embedded systems.

Contents

1 Introduction **4**

2 Background **6**
2.1 The previous assignment: Traal-Rover 6
2.2 The new tools 8
2.3 Statecharts 12

3 Prior Work **16**
3.1 TurtleBot3 in research 16
3.2 TurtleBot3 in education 17

4 Method and Approach **18**

5 Development **20**
5.1 The new Traal-rover assignment 20
5.2 The use of Gazebo during development 22
5.3 The ROS 2 interface and connecting to SCT 23
5.4 The statechart solution to the assignment 24

6 Results **26**
6.1 Capabilities of the new statechart tools 26
6.2 The final performance 29

7 Conclusion and Discussion **33**
7.1 Q1 33
7.2 Q2 33
7.3 Q3 34
7.4 Additional Information 34
7.5 The Main Question 35

8 Future work **36**

A Zoom in on the final Statecharts **39**

Chapter 1

Introduction

In recent years, robotics has gained considerable attention for numerous applications. As such, there is also an increased interest in education for robotics [7]. Currently, there are several robotic platforms that can be used in education. This research focuses on two of these robots, the Lego Mindstorm EV3 and mainly the TurtleBot3.

The EV3 is a platform designed for hands on education[1]. It is a commercial robot and comes with visual programming software. Additionally, it can also be programmed with programming languages. A main feature of the robot is that it is modular. The EV3 can be edited to any configuration necessary for the type of education. The main target audience of the EV3 would be children in primary or secondary school. On the contrary, the TurtleBot3[2] is an open source robot that can be programmed using the Robot Operating System (ROS)[9]. It has a focus group of high-school and up. Lastly, the TurtleBot3 is also modular and can thus also be edeited to different configurations. [7]

A course at the University of Amsterdam (UvA), Embedded software and systems[3] (ESS), currently uses the Lego Mindstorm EV3 for one of the assignments in the course. However, students ran into inaccurate sensors on the EV3. Therefore, students needed to spend time refining parameters, which was not part of the intended learning outcomes of the course. The goal of the assignment was to work together in a team to develop software that satisfies its requirements on a given embedded platform. Thus, a replacement was necessary, such that the students could focus on the core contents of the course. TurtleBot3 may be the solution here.

The development of this assignment was done using statecharts[3]. Statecharts are a visual formalism that can be used to represent how a complex system works. (In Chapter 2.3 the technical details will be explained.) In the case of the assignment in the ESS

[1] https://education.lego.com/en-us/products/lego-mindstorms-education-ev3-core-set/5003400

[2] http://www.robotis.us/turtlebot-3/

[3] https://studiegids.uva.nl/xmlpages/page/2019-2020-en/search-course/course/73512

course, Mathworks Stateflow[4] was used. However, ROS is not usually programmed using Matlab, the programming language Mathworks is based on. There is a package for Matlab that enables ROS support. But, students had some problems with the Mathworks software. Thus, YAKINDU Statechart Tools (SCT)[5] is used instead to enable the TurtleBot3 to be used as a replacement. SCT does not have ROS support in the base product. However contrary to Stateflow, there is a code generator for Python. This generated code can be used to interact with ROS. Additionally, SCT's code generator can be customised or a new one can be developed by the user, if necessary. ROS does have a package called SMACH that can be used to develop hierarchical state machines. However, this does not support the statecharts formalism.

The TurtleBot3 needs to be tested if it can be used for education in this manner. Therefore, the primary research question of this thesis will be : Can TurtleBot3 be used to teach application programming for embedded systems using Statecharts? To be able to answer this, three sub questions are used:

Q1: Can the new tools, TurtleBot3 and SCT, serve the same educational purpose as the previous tools for the assignment in the ESS course?

Q2: Can problems with the previous assignment be solved by using the new tools?

Q3: What is a suitable level of abstraction for education using the TurtleBot3?

The first sub question, Q1, is the baseline of what needs to be achieved for the TurtleBot3 to be used in this context. This will automatically fail if it turns out a statechart tool cannot be connected to the TurtleBot3. Ideally Q2 is also true as the TurtleBot3 is not just meant as a replacement, but also as an improvement. Q3 is catered towards the technical details. To teach application programming for embedded systems, a suitable level of abstraction needs to be used to enable students to focus on the materials of the course. Students should not have to spend much time with things that are not part of the course.

This thesis starts with a background to the assignment in the ESS course and all the tools mentioned up until now. Then, in Chapter 2.3 there is a basic explanation of the formalism of statecharts. Next, the applications of the TurtleBot3 in research is discussed in Chapter 3 followed by the robot's use in education. Afterwards, in Chapter 4, a method is proposed to give answer to all the questions previously stated. Then, the development of all necessary tools to prove the questions is discussed in Chapter 5. next, the results is discussed in Chapter 6, followed by the conclusion to the questions in Chapter 7. Lastly, in Chapter 8, some future work is proposed.

[4]https://nl.mathworks.com/products/stateflow.html

[5]https://www.itemis.com/en/yakindu/state-machine/

Chapter 2

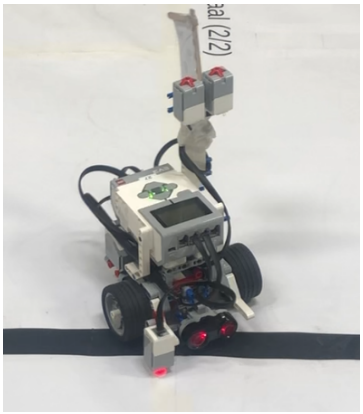
Background

In this chapter, the previous assignment is outlined as well as the tools used for it. Then, the new tools are established. Lastly, an introduction will be given to the formalism of statecharts.

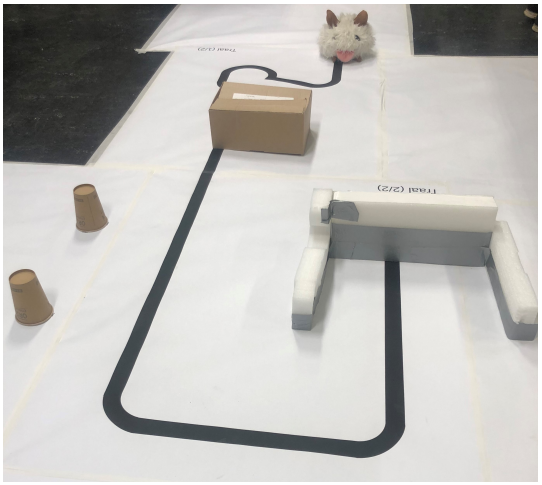
2.1 The previous assignment: Traal-Rover

As the previous assignment is used as the base criteria, it is important to know what tools were used and what the assignment consisted of. The tools for this assignment were the Lego EV3 Mindstorm and Mathworks Stateflow. Firstly, the configuration of the EV3, that can be seen in figure 2.1a, was as follows. in terms of sensors, it had 2 touch sensors facing up for manual control, a color sensor facing downwards, and an ultrasonic sensor facing forward. For movement, it had a motor for the left and the right wheel, and a motor to raise a flag. These motors also have sensors to measure their rotation. Secondly, Mathworks Stateflow was used as the visual interface to develop statecharts in. It would directly control the EV3 using Simulink. Further details of the tools will not be used for comparison and are thus not important.

The previous assignment is used as a base for a new assignment that the TurtleBot3 is supposed to complete. The previous assignment was called Traal-Rover, in which students had to make an embedded application using statecharts for the EV3 to explore the planet Traal by following a path and logging what it encounters. Furthermore students had to work in groups, as the goal of the assignment was to work together in a team to develop software that satisfies its requirements on a given embedded platform. Additionally, it was used to teach statecharts and how to work with them in a practical manner. These were the main educational purposes and the TurtleBot3 burger should at least serve this purpose to be used in the course.



(a) The EV3 Configuration



(b) The physical model

Figure 2.1: Picture of the test setup for the Traal-Rover assignment for Embedded software and systems 2019/2020 academic year

The Traal-Rover assignment consisted of 6 steps, which needed to be completed by five components. The course that needs to be completed can be seen in figure 2.1b. The first component is *manual control*. This component makes use of buttons and timing to control the robot's movement. It is meant to complete Step 1, which is driving manually to an indicated area. Additionally, if the EV3 fails any step, manual control can be used to bring the robot to the next step. After the Traal-Rover is driven to the indicated area, Step 2 is to autonomously drive forward until it encounters a line on the ground, also known as the path. It needs to turn in the correct direction and align with the path. Then, it should follow the line. This is Step 3, which will mostly be completed by the second component. This is the *line following* component, it makes use of the color sensor to determine where the line is. These two components complete the base program of the robot.

Furthermore, the next three components add additional functionality and each have their own step. The third component is *obstacle avoidance*. Whenever there is an obstacle on the path, the robot should avoid it along its left side. An obstacle can be detected using the ultrasonic sensor in the front. The amount of obstacles also needs to be counted. For the next component, further use is made of the ultrasonic sensor. The *tree counting* component is used to count trees that are situated on the right side of the path. These trees in the form of cups are always in a range of 30 cm from the line. The last component, as well as the final step, is *parking*. When the Traal-Rover reaches the end of the line, marked by walls on the left, right and front, it should stand still. When the robot is stopped, it has to display the amount of obstacles and trees encountered on the screen of the EV3. Lastly, it has to beep for one second and raise a flag.

This assignment was done by students in the first semester of the 2019/2020 academic year. After courses at the UvA, students get the possibility to deliver feedback on a course and its individual parts. For each assignment in the course, the students rated it on whether the assignment helped understanding the subject matter of the course, as well as some other criteria. The rating for this is on a 5 step scale from strongly disagree to strongly agree. This rating can be used as an indication of the instructive and thus educative quality of the assignment. In this criterion, it is still rated positively with an 3.6/5, but it has the lowest of the assignments in the ESS course. Which is unfortunate, because it is the most time-consuming assignment. This lower score is further elaborated on in the open question at the end of the form. Students and teacher raised the following problems with the Traal-rover assignment[1]:

- The Traal-rover assignment does not parallelize well. This results in that groups often split up, some people working on the Traal-rover assignment, while others are working on other parts of the course.
- Students spent much of their time optimizing sensor settings, instead of designing logic. This happened due to the quality of the sensors for the EV3 being lower than desired.
- Matlab/Mathworks Stateflow had crashed for students, both on Debian and macOS. Additionally, Matlab is not often used as the main programming language of a course at the UvA.

These problems are not implicitly negative. For example, optimizing settings reflects realistic challenges in the embedded domain, but at a certain point there is nothing more to learn from this. Thus, if it is possible to replace this aspect of the project with something more challenging or rewarding for the students, they can get more value out of the time they spend on the course. Additionally for Mathworks Stateflow, because it crashes and students do not often work with Matlab at the UvA, it is not the best solution for this course. However, Mathworks Simulink and Stateflow is still the most commonly used modeling software and thus would give students appropriate experience for the domain [4].

2.2 The new tools

The new tools are meant to solve the problems with the previous assignment and improve its educational quality. However the new tools should at least serve the same educational purpose as the EV3 and Mathworks Stateflow did. The new tools are the

[1] According to the 2019/2020 academic year student feedback document and the teachers of the course Embedded Software and Systems

TurtleBot3 Burger and YAKINDU Statechart Tools (YST). With the TurtleBot3, extra tools are included. The operating system is the Robot Operating System (ROS), a set of software libraries and tools for building robot applications, and a simulation model for the TurtleBot is provided that can be used in Gazebo[2].

The TurtleBot3 comes in three forms, the burger, waffle and waffle pi. As cost is a consideration in education, the cheapest of the three was chosen. This is the TurtleBot3 burger, which has a Laser Distance Sensor(LDS) and an Inertial Measurement Unit (IMU) as sensors. The burger version does not have a camera, but the waffle versions do. As motors it has two actuators for the wheels and these can give information about odometry.[3] The last main feature is that TurtleBot is a ROS [9] standard platform. ROS is a middleware based on a publisher/subscriber mechanism. Communication is done through a network of nodes. These nodes publish or subscribe to topics, where publishers publish a message and subscribers read that message using callbacks[4]. This mechanism can be seen in figure 2.2. It also provides the first level of abstraction for Q3, as there is no direct interaction with the TurtleBot3's actuators. The interaction here is with the message data types ROS uses for topics. For example speed and rotation are set by sending a Twist message to the ROS velocity node and not by controlling each actuator separately. Thus this is the lowest level of abstraction that can be achieved using the TrutleBot3.

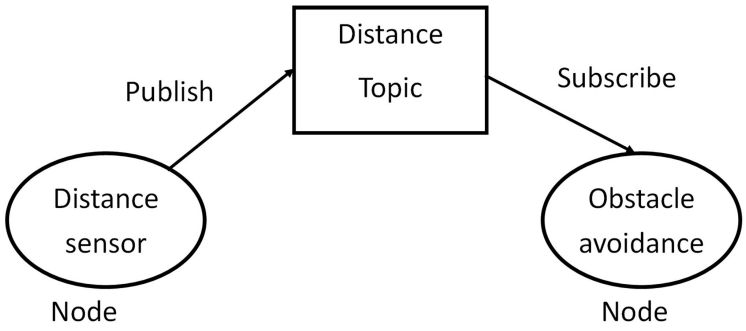


Figure 2.2: ROS nodes and topics

In the case of TurtleBot3, different versions of ROS, ROS 1 and ROS 2, are supported. The versions are slightly different in structure, as can be seen in Figure 2.3. Thus, a decision needs to be made which one to use. ROS 2 is a new version of ROS with the goal of adapting to changes in the robotics community, while still leveraging what is great about ROS 1. However, currently ROS 1 is still used the most, as can be

[2]http://gazebosim.org/

[3]https://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/#specifications

[4]https://index.ros.org/doc/ros2/Concepts/#conceptshome

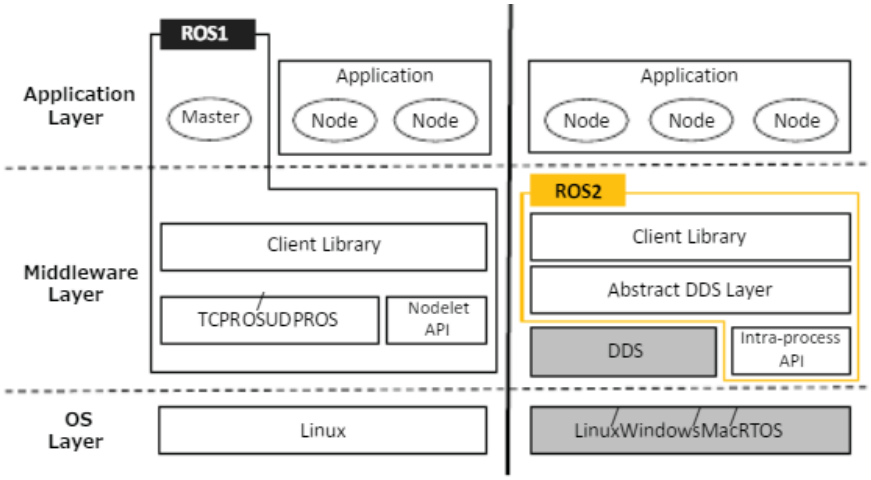


Figure 2.3: ROS and ROS2 architecture overview [5]

seen in Figure 2.4. As of the time of writing both ROS 1 and ROS 2 are supported, but development will slowly go more towards ROS 2. This means that using ROS 2 is more future proof as education should use the newest versions of industry standard programs as much as possible. However, there is less reference material available for ROS 2, thus students might have trouble solving problems they encounter. However, this can also be used to increase the amount of reference material by students asking questions on fora. Lastly, ROS 2 seems more promising for use in real-time embedded systems than ROS 1 according to Yuya Maruyama et al [5]. Thus, it is better for the type of application development the TurtleBot3 will be used for in this thesis. In conclusion, ROS 2 was chosen for this project as it is best to educate students on this platform for embedded systems.

ROS 2, contrary to ROS 1, does not have a master node, thus removing the single point of failure. ROS 2 does not specify a threading model for an application. Instead, publishing and callbacks for subscribers is done by spinning a node. Spinning can be done in three ways in Python. Firstly the spin() function, which continues until the node is shutdown and breaks program flow fully. The second option is spin_until_future_complete(), which executes until a provided future object is completed. Lastly, there is the spin_once() function, which will execute one callback, unless a timeout expires before a callback is received[5]. In the case of this thesis, spin_once() will be used to line up the execution of ROS and the statecharts. ROS 2 is all-together difficult to master, so for education for embedded systems using statecharts, this will need to be further abstracted for students to work with.

The simulation tools that will be used for ease of development, when there is little time available with the physical robot, is Gazebo. Gazebo provides a 3D simulation

[5]http://docs.ros2.org/latest/api/rclpy/api/init_shutdown.html

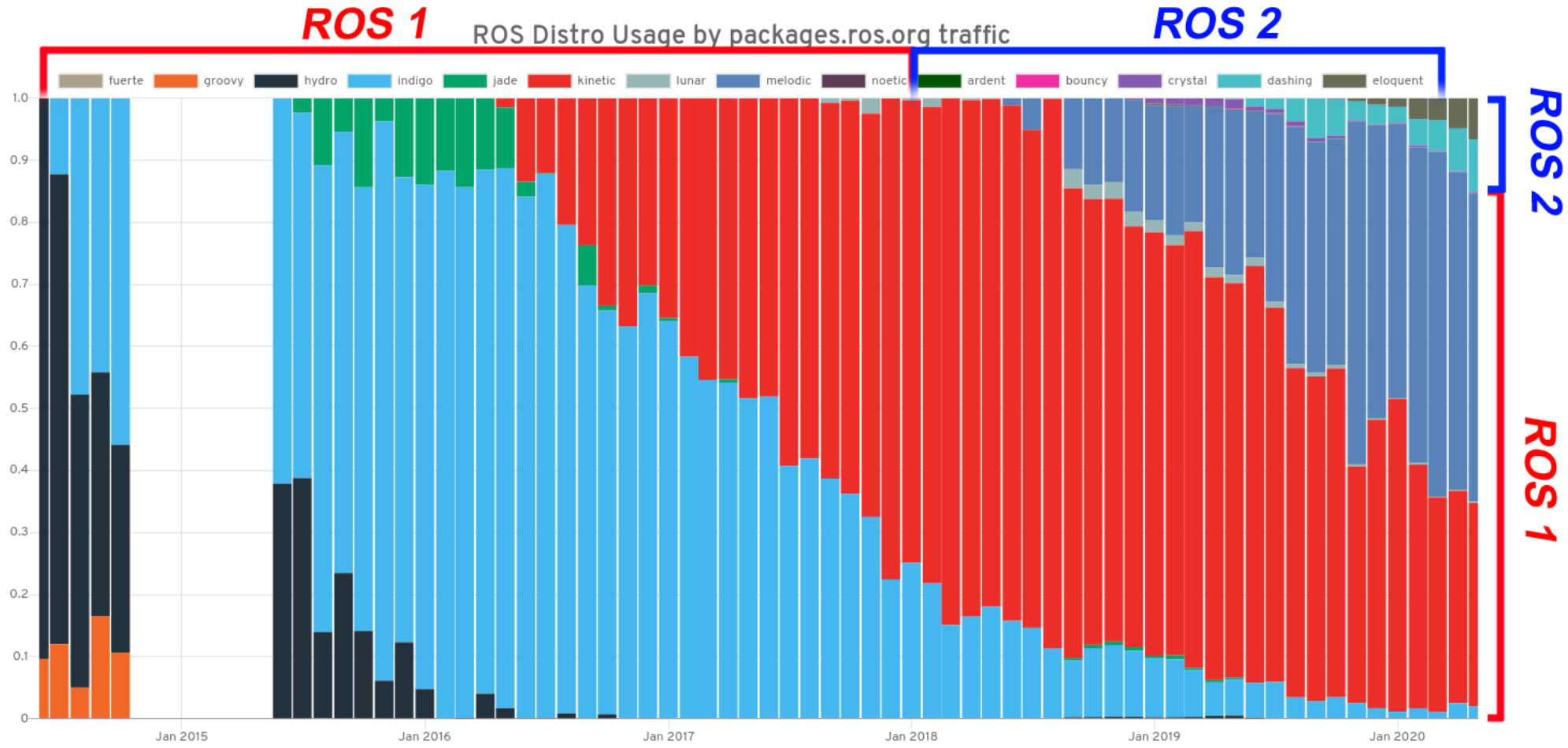


Figure 2.4: The distribution of use of ROS packages.
Fuerte till noetic is ROS 1, ardent till eloquent is ROS 2.
Source: https://metrics.ros.org/packages_rosdistro.html

environment in which models can be added. For the TurtleBot3, this model is available online and can be used in Gazebo with ROS fully working for all sensors. During the time of this project Louis van Zutphen was testing the fidelity of this simulation at the UvA. Any assumption about the fidelity in this report were based on results he gathered [10]. Gazebo also has the option to import models made in other modelling tools. For this project Blender[6] will be used to create the complex models.

The last of the new tools is YAKINDU Statechart Tools (SCT), an Eclipse based open source visual interface to develop statecharts. This tool was chosen because it has a code generation tool that can generate Python and C++ code, which are the supported programming languages for ROS. Additionally, Eclipse based tools are the second most used modelling software in the embedded domain [4]. Thus, it is still useful experience in this domain, just as Mathworks Stateflow and Simulink. YAKINDU is currently in the beta of SCT 4.0. where the Python code generator is a standard part of the product, whereas it was previously a YAKINDU LABS[7] feature. Because Python will be used to connect with ROS, SCT 4.0 beta was used for this project. While SCT can automatically generate Python code, it does not have automatic code generation for ROS. Thus, extra code will have to be written to pass data between the statecharts and the robot.

2.3 Statecharts

Statecharts are a visual formalism for complex systems that describes states and transitions in a modular fashion. Contrary to other modelling formalisms, statecharts are meant to be part of a system and not just a visual representation. The visual formalism of statecharts is meant enable clustering, orthogonality and refinement [3]. In Figure 2.5, a statechart for two traffic lights can be seen. The set up for these traffic lights is one road with a crosswalk. The traffic lights are meant to provide save crossing for pedestrians. This section will give information about the components of statecharts and then do a walkthrough of the traffic light statechart.

The Traffic light statechart is a YAKINDU example and it displays the ability of statecharts to 'zoom' in to different layers of abstraction. For example, the most abstract layer in this example is whether the traffic light is on or off. However, in a less abstracted layers, it is depicted what actions need to be taken when pedestrian is waiting to cross the road. Additionally, the details of what lights need to be on are in a less abstracted layer.

[6] https://www.blender.org/

[7] https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/lab_projects

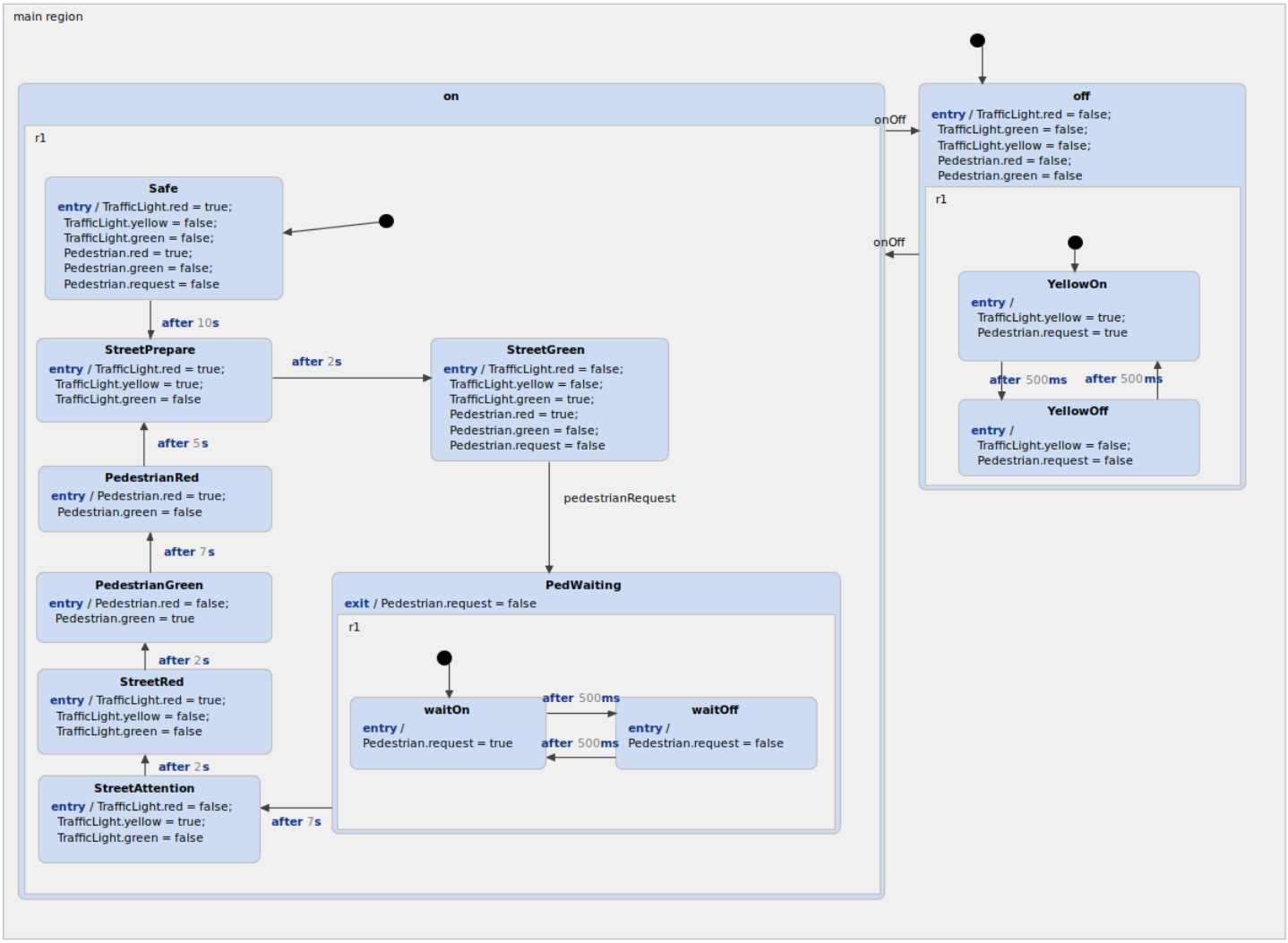


Figure 2.5: The statechart to control a traffic light from YAKINDU SCT examples

The flow in this example is based on *timed transitions* and *event based transitions*. Transitions are displayed using arrows, and the type is displayed using the text along the arrows. In this statechart, there are two events that can trigger transitions: onOff and pedestrianRequest. On top off timed transitions and event transitions, statecharts can also use *conditional transitions*. For example, instead of just an event, the pedestrian-Request can be a button that records the length of time it is pressed. If it is not released for longer than one second, beeps would start so blind people know if the light is on green.

The light changes in this statechart happen upon entry or exit of certain states. This is done by using the entry and exit actions of a state. States are represented by the

blue rectangles. The two types of states used in this example are *normal states* and *composite states*. Composite states are states that contain different states in an area inside them, such as the on and the off state. The last possible state, which is not used in this example, is an *orthogonal state*. This is a composite state where more than one internal statechart exist in different areas. It is used to compactly represent a state space by allowing multiple statecharts to be active at the same time. It can be seen in Figure 2.6. In this statechart, a state in area A (C or D) as well as a state in area B (E, F or G) are active at the same time.

Every area contains an entry point, which signifies where the internal statechart starts. This is represented by a black dot with a transition to the starting state. If the start is not the same for every entry, conditional transitions or histories can be used to choose what state to start in. The history can be used to go to the last state that was active at the level or lower levels of the entry. A history used to go to the last active state before deactivation on the same level as the history is a *shallow history*. Additionally a history that can go to lower levels as well is called a *deep history*. In Figure 2.7a an example is given with a shallow history. When this composite state is entered, the shallow history activates either state A or state B, because these are on the same level. This is dependant on which of these states was the last to be active. However, if this history would be a deep history, of which the icon can be seen in Figure 2.7b, there is a difference. Instead of deciding between A and B to be active, the history takes lower levels into account. Thus, the history activates either state C, D, E or F, as these are the lowest level states. Say that before deactivation the composite state B was active, in B's area E was the last active state. If a shallow history is used upon the next entry, B will be activated, which will activate F. However, if a deep history is used, instead of F, E will be activated.

Knowing these formalisms, the flow of the statecharts in Figure 2.5 can now be followed. The main entry is above the off state, when the program is executed the off state will thus be entered. When the *off* state is entered, the program will switch between the YellowOn and YellowOff state every 500 ms using a timed transition. This will cause the yellow light of the traffic light to flicker. If the onOff event is triggered, the state *off* will be deactivated and the state *on* will be activated, using an event based transition. Upon entry in the composite state *on*, first all traffic lights will be red. Then the traffic light for vehicles will have the yellow light on as well, followed by only the green light being on. The statecharts will remain in this state until a pedestrian comes up and pushes a button to cause the pedestrianRequest event. At which point the statecharts will follow the cycle from the PedWaiting composite state, until it comes back to the state second from the top, the StreetPrepare state. Then the cycle will restart and the pedestrian light will be red, and the vehicle traffic light will be green again. If the onOff event is triggered again the *on* state and all its sub-states will deactivate. Then the *off* state is once again activated.

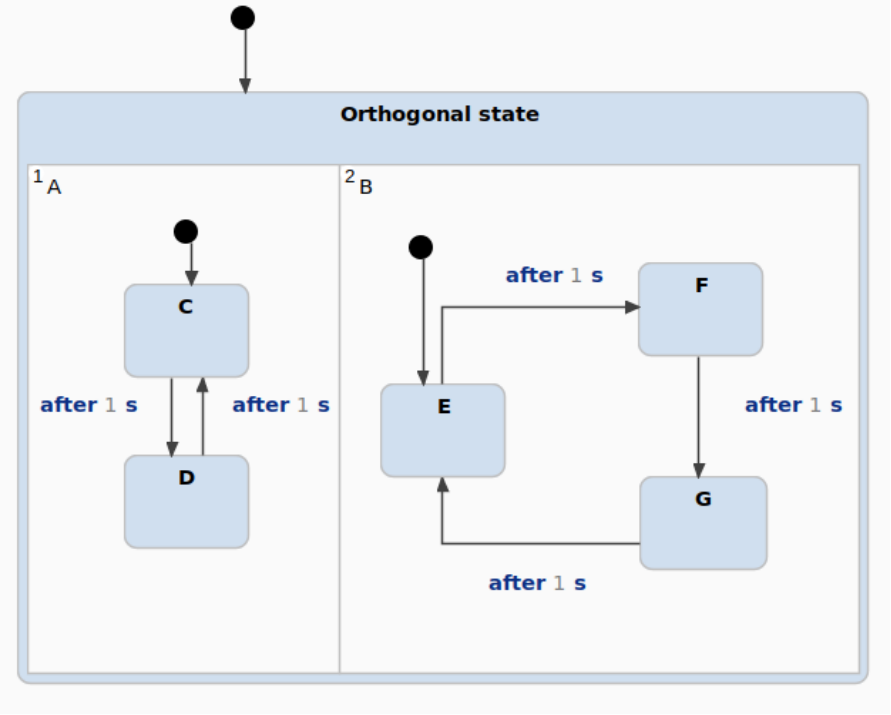
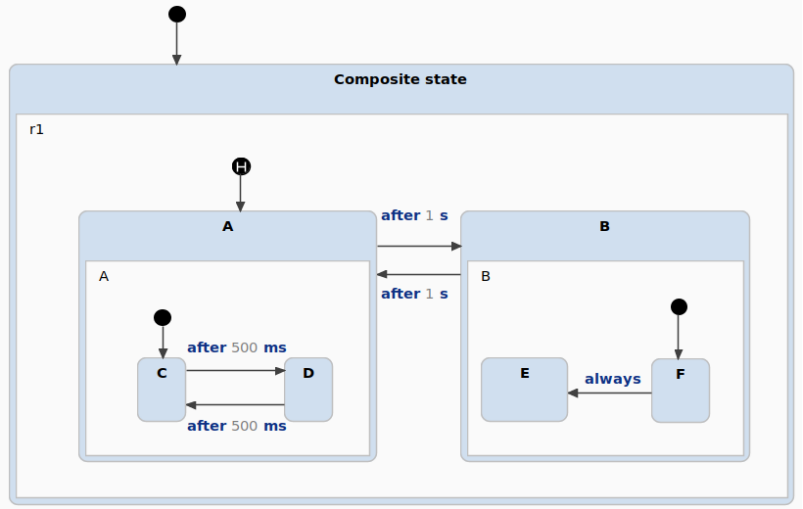
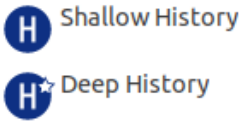


Figure 2.6: An orthogonal state in YAKINDU SCT



(a) A state containing a shallow history



(b) The symbols that can be used for histories

Figure 2.7: Histories in YAKINDU SCT

Chapter 3

Prior Work

In this chapter, research using and research on the TurtleBot3 will be discussed. In the first section, applications of the TurtleBot3 in research will be discussed. This will sketch a picture whether the sensors are fit for the application of this thesis, as is necessary to solve one of the problems. Additionally, it shows the area of research the TurtleBot3 is normally used in. Then, previous research on using the TurtleBot3 for education will be discussed.

3.1 TurtleBot3 in research

The TurtleBot3 was released in May 2017, since then it has seen some use in research. The main area of research it is used in is autonomous vehicle navigation. It is, for example, used by researchers to compare different types of navigation [8]. Additionally, it can be used for Simultaneous Localization and Mapping to map a ventilation system [6]. This shows that the TurtleBot3 is considered by researchers to have sufficient quality to be used for their research. Thus, the quality of the sensors and actuators should also be sufficient to use in education in the context of autonomous navigation. Additionally, ROBOTIS has videos online that display the autonomous navigation qualities using ROS packages and more[1], such as the modularity of the TurtleBot3. In the context of this thesis, the TurtleBot3 will be used for autonomous navigation. As this is the focus within research using the robot, it also fits well in the application the TurtleBot3 is used for in this research.

[1] https://www.youtube.com/playlist?list=PLRG6WP3c31_XI3wlvHlx2Mp8BYqgqDURU

3.2 TurtleBot3 in education

The TurtleBot3 has previously been tested on educational quality by researchers at the KU Leuven [1]. Their motivation was to have hands on learning for a particular course, and the TurtleBot3 stood out from the other options. In their case, students also did not directly interact with ROS, just as the assignment in this research will be. However, instead of working with statecharts, they worked with MATLAB code. A custom class was created to abstract messages from ROS that the students could interact with. For a course on embedded systems, students used this to create a PID controller. Students that followed this course gave positive feedback, thus the TurtleBot3 proved to work well for this setting. However, the project was more challenging than estimated, mainly in the design. For this project, the extra abstraction layer Statecharts provide might be able to take away some of the challenging aspects of the design. Thus, students could then focus more on the application development.

The new tools are further advantageous in education because they are all *open source*. The experience of working with open source tools is inherently beneficial. Open source tools used in the industry give useful experience for students. It teaches them to work with large and complex software. However, in contrary to purpose made tools by teachers or familiar programming languages, there is a downside. Teaching staff will need to provide adequate support to learn these tools as students are often unfamiliar with them. [2]

Chapter 4

Method and Approach

To start testing the TurtleBot3, three steps of development need to be completed. The first is to conceptually redesign the Traal-rover assignment to work with the new tools. Secondly, a ROS2 interface is implemented. Thirdly, a statechart solution is created for the assignment with the new tools.

The redesign of the previous assignment is used to establish if the new tools can serve the same educational purpose, as required by Q1. As there are different sensors and different development tools to work with, the assignment will need to have a different test setup. However, it should maintain the goals of the Traal-rover assignment. Thus, if the TurtleBot3 is able to complete the Traal-rover assignment using a statechart solution, the new tools can serve the same educational purpose, as long as the capabilities of the robot and the statechart tools are the same or better.

To have the tools work together a ROS2 interface needs to be developed and implemented. This interface should be connected to the generated interface from SCT, a visual representation of this can be seen in Figure 4.1. With this, there will be two new layers of abstraction, that of the ROS 2 interface and that of what data is passed to the SCT. The data that is passed to SCT determines what level of abstraction the students are usually working on. Additionally, the ROS2 interface determines what level of abstraction students could work with if they would look at the code. This step should determine if a statechart tool can be connected to ROS 2, which is the minimum needed to start testing for all questions. Additionally developing this should give insight on what the minimum level of abstraction is at each layer.

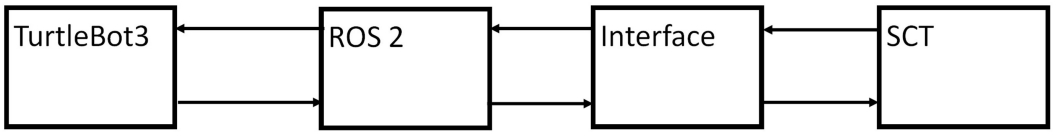


Figure 4.1: This displays the connection the interface will establish

Lastly, the statechart solution for the new Traal-rover assignment will be made. This will be done at the level of abstraction determined by what data is passed to SCT. By completing this and working with the new tools as a student would, it can be determined if the new tools solve the problems of the previous assignment. While developing this, the level of abstraction will be further refined, such that students can get all data they need. This will help answer Q3.

During these stages of development the tools can be further analysed by using them. This should give further insight in the capabilities of the robot. Additionally solutions to the problems with the previous assignment will be searched during development, thus answering Q2. By working together on the robot with Louis van Zutphen while he is testing the fidelity of the simulation [10], further information can be gathered about how to work with the new tools as a group. Then, to complete answers to all sub questions, the robot will be tested if it can complete the assignment. This will give the final answer to if TurtleBot3 and YAKINDU SCT can be used for education of embedded systems in the context of statecharts.

Chapter 5

Development

This chapter will describe the steps proposed in the method. It will start with the redesign of the Traal-rover assignment. Then, the use of Gazebo for development will be elaborated. Afterwards, the programs connecting ROS 2 and SCT will be discussed. Lastly, an overview will be given of the statecharts used to complete the new assignment.

5.1 The new Traal-rover assignment

The first step of development was to recreate the Traal-rover assignment. The main focus was to retain the same challenges, but with the new tools. The largest hurdle here is the difference in sensors on the robot. The TurtleBot3 Burger has just one sensor that can relay information about the world around it, while the configuration of the EV3 had several. Thus, all the capabilities of the sensors from the EV3 need to be handled by a single sensor, or other solutions the new tools bring. To do this, as much data from the laser sensor as possible needs to be used. The laser sensor mainly gives data for two things: The distance to and the intensity of a surface. The distance will still be used in the same manner as the ultrasonic on the EV3, however now at all 360 angles. The intensity values will be used as a replacement of the color sensor. Using the intensities the robot can differentiate between materials such as reflective tape and cardboard [10]. However, the intensity is also affected by the angle the laser hits the surface in, thus the new setup needs walls that are perpendicular to the ground to use the intensity values in this manner.

Knowing what the robot can do, the following will be the replacement for the 6 steps previously mentioned in Section 2.1, while still retaining the five components and their respective challenges. The first component is still *manual control*. However, because the Burger has no buttons on the robot, the way of interacting needed to be changed. To work on the TurtleBot3, secure shell is used to connect remotely. Thus, the terminal to

execute code can also be run remotely. This enables the use of the keyboard to manually control the robot. Therefore, instead of buttons, the robot will now be controlled using the keyboard. The statechart implementation for this should have different behaviors based on what keystroke is used. This will still complete Step 1, driving manually to an indicated area. Additionally, it can still be used to manually drive if the robot fails a step.

From Step 2 onwards, the test setup will be changed. After all, following a line is no longer possible without a color sensor. Hence, the robot will no longer be following a line, it will follow a wall instead. The robot will need to stay equidistant from this wall at 30 cm. This will be the new way to define the path, which will be 30 cm left of the wall. This means that Step 2 will still be driving forward until the path is encountered, then turning in the correct direction to align with the path. Because of this change, the second component used for Step 3 will no longer be line following. From now on it will be called *wall following*. Nevertheless Step 3 will remain the same, as it is to follow the path. This can then be achieved by using the distance sensor to continuously stay at the same distance of the wall. This will again complete the base program of the robot.

The third component will once again be *obstacle avoidance*. Previously this was done by checking with the ultrasonic sensor if something was on the path using distance. However, if only distance is used, the TurtleBot3 will be incapable of distinguishing between an obstacle and a wall. Obstacles will thus have to be marked in an additional way. This is were reflective tape can be used to differentiate an obstacle from the wall using intensity values. The rest of this component will remain the same. Obstacles still need to be counted and need to be avoided on the left side. Avoiding the object on the left side is even more important in this setup, as the rover would crash into the wall if it tried to go right.

For the fourth component, intensity values will continue to be used. The tree counting component can no longer happen in the way it previously did. After all, the right side of the path now has a wall standing in the place where the trees would previously be. So instead this component will be *gem counting*. Pieces of reflective tape will be affixed to the wall. These will represent the gems and need to be counted. This makes further use of the higher intensity values of reflective tape. The last component, *the parking component*, at first glance seems to not need much change to the physical model. However, in testing, the floor area the model needs is larger than the EV3 because of the new definition of the path. Thus, to save space, the last step now reads as follows: When the robot is surrounded on the right and front with reflective tape, stop and display the number of trees and obstacles. Instead of raising a flag and displaying on the screen of the EV3, for the TurtleBot3 this will be printed in the terminal used to execute the program. In Figure 6.4 the new model can be seen.

5.2 The use of Gazebo during development

During development, there was no continuous access to the TurtleBot3 Burger or a room of sufficient size to have a physical model. Thus, Gazebo was used for most of the following stages of development. The largest challenge here was how to correctly handle the intensity values as Gazebo handles these different than the real robot [10]. In Gazebo the intensity is just set to one value per object and does not change based on distance. In the real world intensity varies a more, because it changes based on distance and has noise. Hence, as realistic intensity values as possible were given to the objects to have as high simulation fidelity as possible. For these tests, several models were created, two of which can be seen in Figure 5.1 and Figure 5.2. The white parts will have an intensity value of 5000 W/m^2 and are considered the wall, all other objects are green. These have an intensity value of 11000 W/m^2.

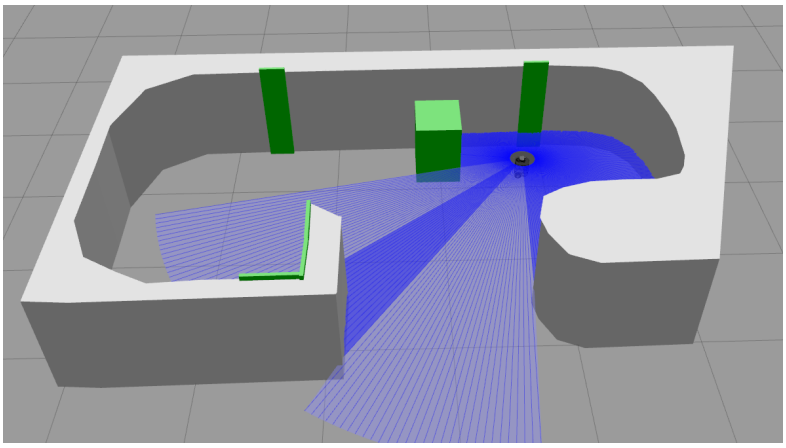


Figure 5.1: A Gazebo model used for development

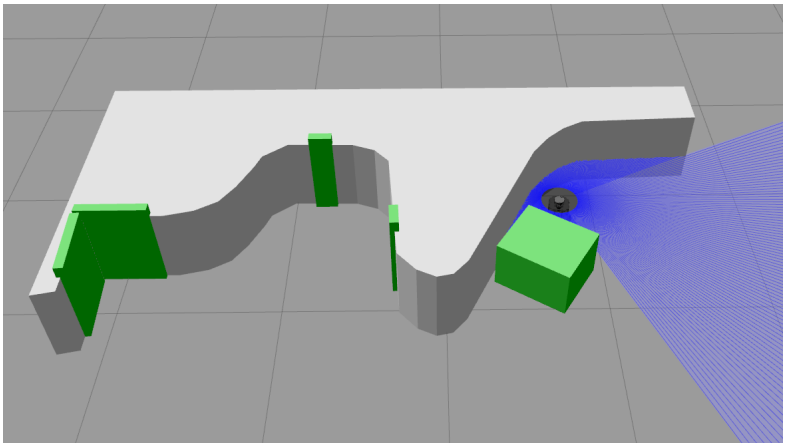


Figure 5.2: Another Gazebo model used for development

5.3 The ROS 2 interface and connecting to SCT

The ROS2 interface is made by making a node class that can communicate in the ROS 2 network. This node will contain the subscriptions for the sensors, and the publisher for the velocity and rotation. This node will filter ROS messages to only the basic data. From the IMU this node contains data about the orientation in quaternions. This node also has the lists of 360 values for the distance and intensity from the LDS. The odometry is not used in this project, as the gyroscope gives all the information necessary about location. Velocity can be published through this node in the x, y and z direction, as well as rotation in all planes. In practice, however, only movement in the x axis and rotation around the z axis will be used, because the robot moves in a horizontal plane and does not have omnidirectional movement. The node is the second layer of abstraction, which is kept as close to the layer of abstraction provided by ROS 2, such that most data can be accessed. However, some data is still lost by reducing the ROS message to only the basic data.

Next to the node class, there is a second class that will connect the TurtleBot node and the SCT generated interface. For SCT complex data types such as lists are not used often. Thus all the data that was in lists will need to be abstracted. For the imu, this means that the quaternions will be made into Euler angles. Hence, for the statecharts the pitch, roll and yaw can be used as data. The laser data will be abstracted in three different manners. Firstly, the interface will contain the data from the lasers at the 0°, 90°, 180°and 270°angles. Secondly, there will be the minimum (min), mean and maximum (max) values of all angles, as well as in what direction the min and max are. Thirdly, the min, mean and max in a range around the 0°(front) and 270°(wall) angles will be available to use for the statecharts. This class will also do some data-preprocessing. For example, zero values from the LDS will be filtered out of the lists,

because it, for example, gives zero value when there is no surface within its working range.

To publish the velocity from SCT to the TurtleBot, the statechart interface will contain a speed and rotation value. These will continuously publish to the Burger as velocity in the x direction and rotation around the z axis respectively. Additionally, this class will also keep track of the keyboard controls for the manual control. When this class is executed, it will continuously access data from both the ROS 2 interface and the statechart interface. It also controls the cycles for both interfaces, every time the ROS node is spun by spin_once(), the statecharts will run a cycle by using YAKINDU's event driven statecharts.

5.4 The statechart solution to the assignment

In Figure 5.3 an overview of the solution to the statecharts can be seen. It is color coded by what step is completed with it. Red is the manual control, yellow is Step 2: driving to the path, green is wall following, purple is obstacle avoidance, pink is gem counting, and lastly grey is parking.

The manual control is made using an orthogonal state with five areas for when w, a, s, d or x are pressed. It switches to autonomous driving when the m key is pressed. These key presses are all event transitions. It goes through the yellow start procedure only the first time a switch is made between manual and autonomous mode.

The autonomous state is once again an orthogonal state, one area for movement and one area for non-movement. In the non-movement area, there is a small statechart that keeps track of the gems. In the movement area, there are two main composite states. The first composite state completes wall following by driving forward and constantly realigning with the wall. The second composite state handles obstacle avoidance and parking. For obstacle avoidance, it is a given that any obstacle is a rectangular cuboid. Thus, moving around an obstacle can be done by turning 90 degrees and then driving past the object several times, until the path is encountered again. Parking is done by moving close to any surface with high intensity. Then, if there is a high intensity on the wall on the TurtleBot3's right it stops and finishes the program.

For more detailed pictures of the singular components, see Appendix A.

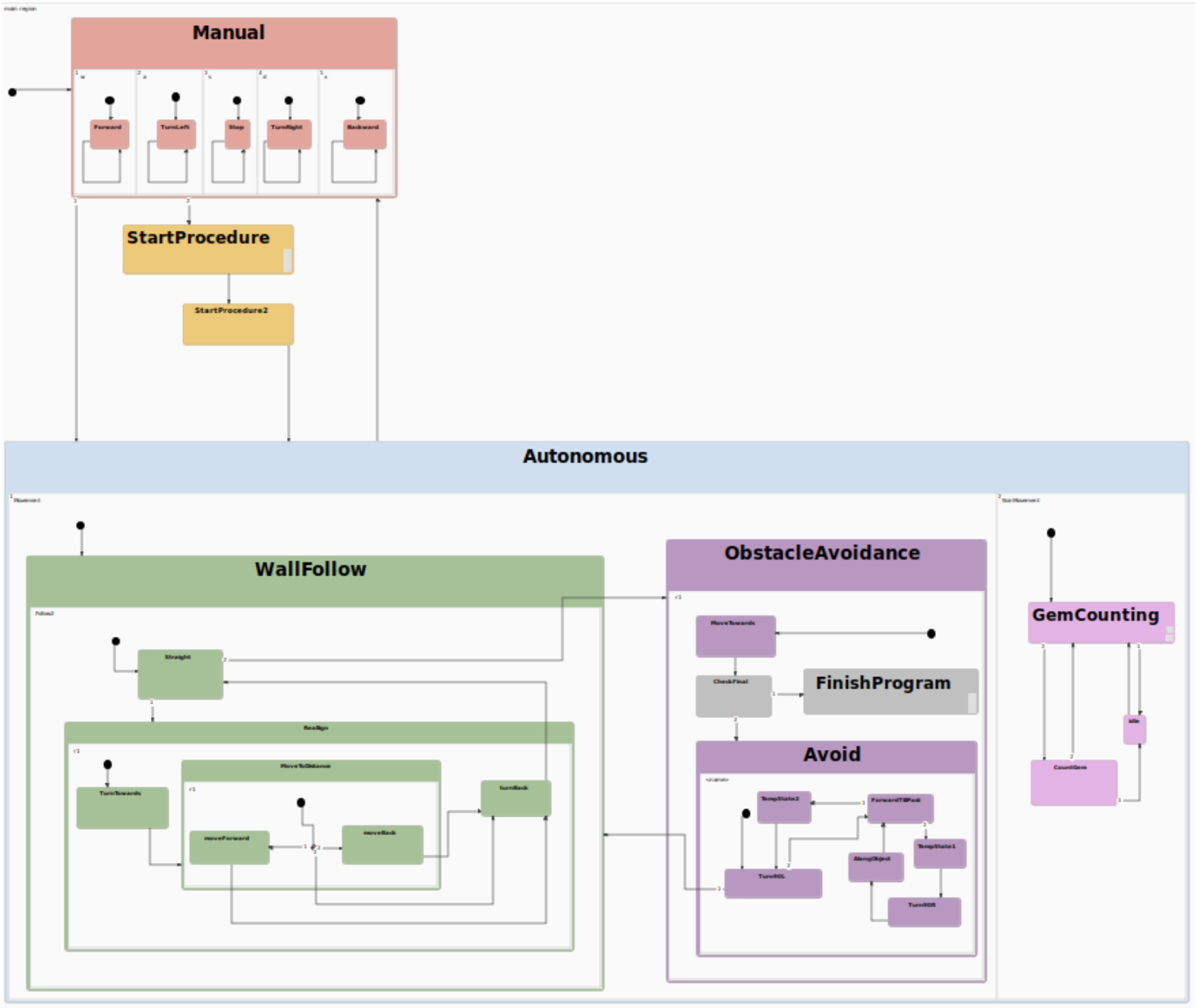


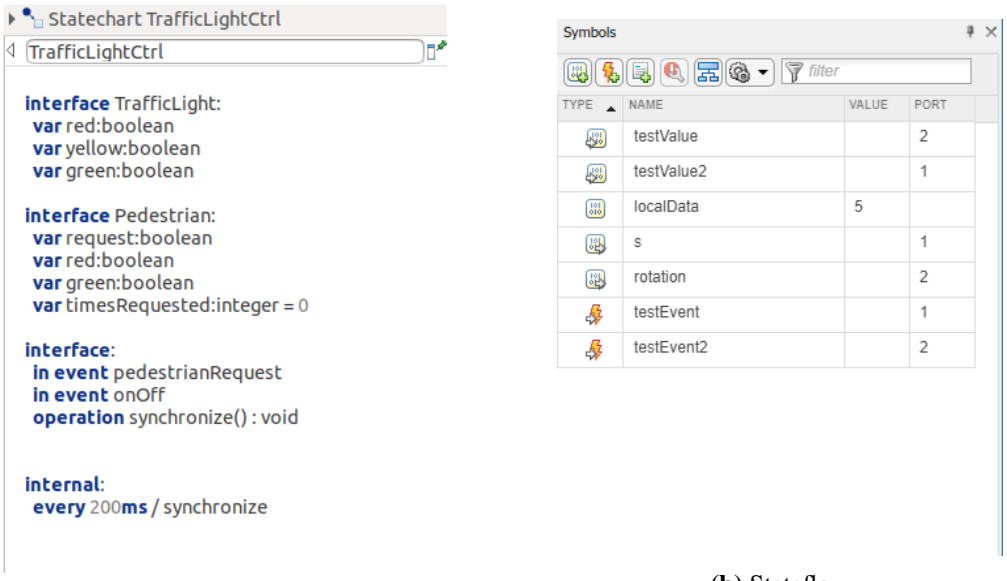
Figure 5.3: An overview of the statechart solution with each step/component colored.

Chapter 6

Results

6.1 Capabilities of the new statechart tools

Working in Stateflow and working in SCT is quite different. Both have access to all states and transitions mentioned in Section 2.3, however the layout of the program and representation of the states is different. The first difference is how the interface works, that programs from outside the statechart and the statechart itself can interact with. In Stateflow, this is represented by a table. However, in YAKINDU this is represented by a code interface. This can be seen in Figure 6.1. In this code interface it is necessary to give the variable type, such as integer, any time a variable is added. The reason the difference in interfaces exists is probably because Stateflow is part of Simulink, which will often be the only program interacting with the statecharts. In contrast SCT is based around generating code for the user to connect to and use in a program. This is further confirmed by that YAKINDU SCT can be directly launched, but to launch Stateflow, first Matlab and Simulink need to be launched.



(a) SCT

(b) Stateflow

Figure 6.1: Variables interface of SCT and Stateflow

The second difference between the two programs is the way states are represented and added. In Simulink all states automatically become a composite state by just placing another state into the state. However, in SCT states can not just be placed within each other. To create a composite state an area in which states can be placed needs to be added to a state first. Thus, the difference between a normal and composite state can be easier identified than in Stateflow. Additionally, in SCT an orthogonal state can be made by adding another area to a composite state. Adding an orthogonal state can also be done directly from the quick menu. An orthogonal state in SCT can be seen in Figure 2.6. In Simulink, creating an orthogonal state is done by going into the top menu to change a state's properties. The properties that switch a state between composite and orthogonal are exclusive and parallel respectively. This changes the visual outline of states within an orthogonal state to have gaps, while states within composite states have continuous lines. These states within the orthogonal states serve the same purpose as an area in SCT. In Figure 6.2 an orthogonal state can be seen. This statechart has the same functionality as the statechart in Figure 2.6.

The last main difference is the way the histories work. In YAKINDU there are two histories, a shallow and a deep history. Stateflow only has functionality for the first of the two, a shallow history. In Figure 6.3 an example of a composite state using a history is shown. This statechart has the same functionality as the statechart that can be seen in Figure 2.7a.

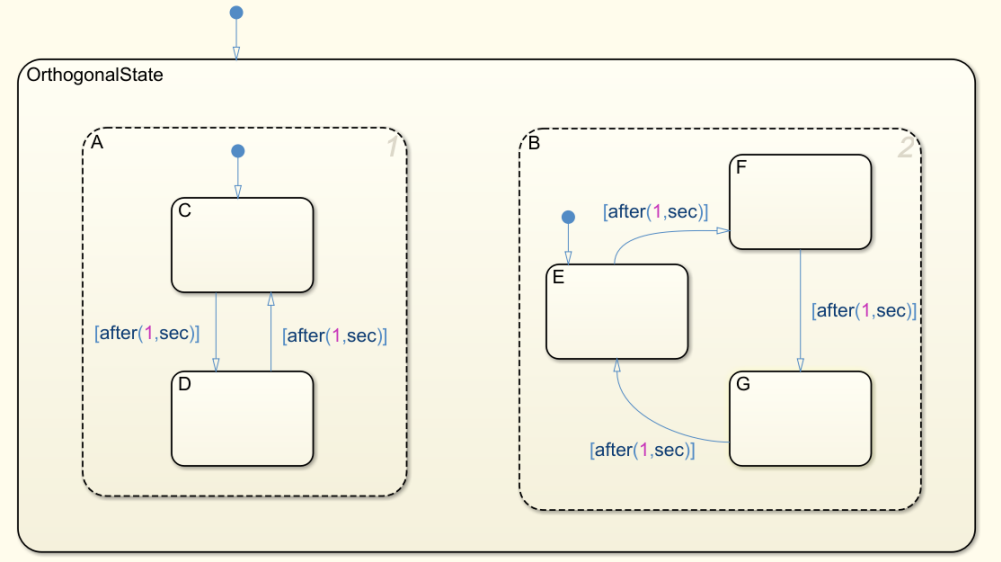


Figure 6.2: Example of an orthogonal states Stateflow

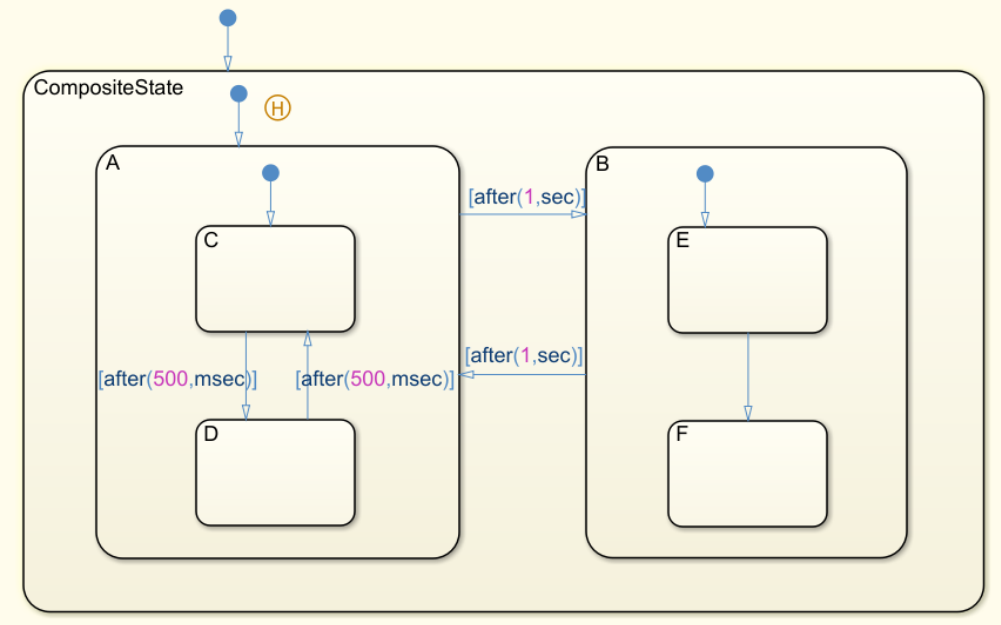


Figure 6.3: Example of a shallow history in Stateflow

Continuing on with YAKINDU SCT in particular. During all development, the tools have not crashed. The visual interface contains all statechart tools that are necessary to complete the assignment, and has more tools if necessary. The code interface that is generated by the Python code generator worked well. Apart from the fact that a relative

import was generated where it should not be used, as the document imported was in the same folder. This resulted in the code not running in a terminal unless the dot of the relative import was removed from the import. However, as the version used is still in beta this will most likely be fixed in the final release. Because ROS was used for the program, the normal Ubuntu terminal was used instead of the inbuilt terminal of SCT. It was easier to keep track of both ROS and the statecharts by doing this. This is possible, because code that could be run in a terminal is generated by SCT.

As mentioned in Chapter 5, SCT did require abstraction of lists. However, because statecharts are used to program the assignment, this is not detrimental to the development of code. Additionally, by having access to the code and the ROS 2 interface, students can always create new variables and different levels of abstraction if that fits their needs.

6.2 The final performance

The TurtleBot3 was tested several times to complete the assignment with the current statecharts on both a physical model and a Gazebo model. In Figure 6.4 the physical model and in figure 6.5 the Gazebo model are displayed. The Gazebo model has the same intensity values for surfaces as the test models used to develop the ROS 2 interface mentioned in Section 5.2. The model contains one object at the start of the path and two gems on the walls, one between the two turns and one between the last turn and the end. The area of the model has a length of 4.5 meters and a width of 2.5 meters.

Figure 6.4: The real world model used for testing

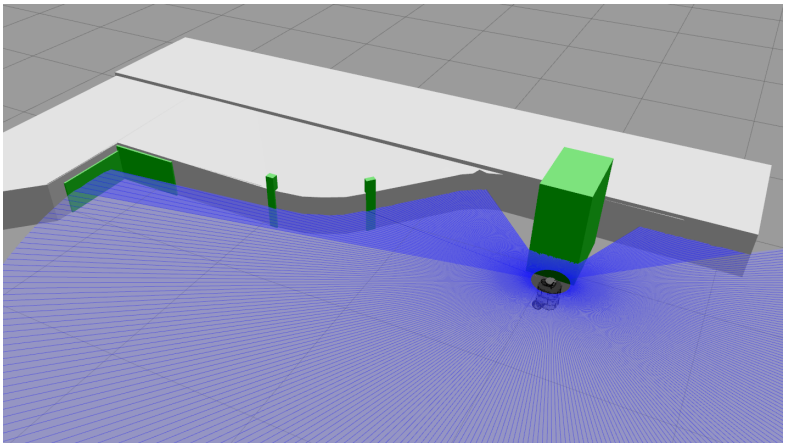


Figure 6.5: The Gazebo model equivalent to the real world model

The TurtleBot3 completed this course several times reliably in the real world, while in Gazebo it had some trouble. In the real world, it consistently completed the course in around 5 minutes and 20 seconds, counted all obstacles and gems correctly and stopped at the parking spot. In Gazebo, however, it could take from 6 minutes to 7 minutes. However, as mentioned before, the Gazebo model was not completely accurate. The TurtleBot3's sensor are quite accurate on shorter distances, while in Gazebo there is a lot of noise [10]. This means that for the TurtleBot3 little fine tuning of threshold values was necessary. However, in Gazebo more fine tuning will be necessary. Figures 6.6 to 6.8 display some of the stages of the robot executing the application.

While testing the new tools, Gazebo had some further troubles. When entering the model editor while the TurtleBot3 model was active, it would fully crash the ROS connected to the model. However, this could be fixed by a restart of the program. It is also important to not use too complex models, as they can get CPU intensive, thus slowing down simulation. In the real world however the application would still have some problems with ROS. Sometimes no data from the IMU would be received by in the first spin and the program would crash. However this can be solved by making a separate node per subscription and waiting for information from all nodes before using the data. This also reduces the risk of data from different sensors that do not belong together used in the same cycle. For example the data from the IMU and the LDS should be from around the same timestamp.

When comparing the statecharts for the new assignment, with the statecharts for the old assignment, the outer levels retain a similar level of abstraction. But when 'zooming' in on the statecharts, the solutions become different for the different tools. However, they solve the same challenges. All levels of abstraction passed to SCT, mentioned in the development, are used in the final statecharts in different situation where they were appropriate.

The main difference between the TurtleBot3 and the EV3 is the level of abstraction

the sensors set on the program. The sensors for the EV3 are all specific for one purpose and give a higher level of abstraction. The sensors of the Burger are used for multiple purposes. This gives the robot less context, making it more general. Therefore the use of certain abstractions given by more sensors is prevented.

As for working with several people on this assignment, during testing two people worked together on the TurtleBot3. This worked well as multiple laptops can remotely connect to the same robot. Additionally, the SCT statecharts can be easily shared across devices by using a cloud code sharing application, such as GitHub[1] and importing and exporting the statecharts.

(a) Start in indicated area of the robot

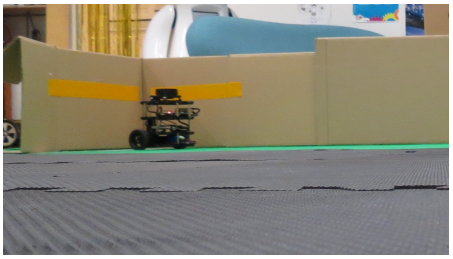
(b) Allignment with the path

Figure 6.6: The second step of the program

Figure 6.7: Avoiding an obstacle

[1]https://github.com/

(a) Counting gems at the tape



(b) The final parking position of the TurtleBot3

Figure 6.8: Counting gems and parking

Chapter 7

Conclusion and Discussion

To conclude, all sub questions need to be answered. However the basic question needs to be answered first: Is it possible to work in a statechart context using the TurtleBot3? The simple answer here is yes, but abstraction is necessary to complete it. This allows us to test for the 3 sub questions set out to answer the main question.

7.1 Q1

First Q1 is answered. This is the base question to prove whether the TurtleBot3 and SCT can be used to teach application programming for embedded systems using statecharts. The first sub question is if the new tools serve the same educational purpose as the previous tools for the Traal-rover assignment. The educational purpose of this assignment is to develop software that satisfies requirements on a given embedded platform in a group. The robot can complete the Traal-assignment in a similar manner as the EV3, with statecharts of similar complexity and the same features solving the same challenges. This addresses the part about developing software that satisfies requirements. Additionally, the assignment needs to be done in a group. As mentioned before, the TurtleBot3 and YAKINDU both allow for sharing between several people. Thus, this satisfies the second part of the educational purpose. This means that the new tools serve the same educational purpose as the previous tools.

7.2 Q2

Q2 was if the new tools could solve problems students and teachers had with the previous assignment. The first problem was that the assignment does not parallelize well, thus groups often split up. It is unclear if the new tools can solve this problem as this remains to be seen when students work on the revised Traal-rover assignment. While

the new tools do allow for collaborating, statecharts can be difficult to simultaneously design in a university setting with larger group without real-time sharing capabilities. Neither Stateflow nor SCT has this capability. If development is only done while there is access to the robot, having real-time sharing would be a requirement for parallelisation. However, as development for the TurtleBot3 can be done in simulation, simultaneously working on the statecharts might not be necessary. Thus, it reduces the problems with parallelisation.

The second problem was that students spent much of their time optimising sensor settings instead of designing logic. This problem can be solved by having more accurate sensors, which the TurtleBot3 has. Additionally the TurtleBot3 has less sensors, thus, requiring less mastering of sensors. Furthermore, the materials of the test setup were chosen in a manner that the intensity values have the least amount of overlap possible. This will be able to limit the amount of adjustments needed to intensity threshold values used to differentiate surfaces.

The last problem was that Stateflow would crash for several students. The new tools did crash from time to time, as mentioned in Section 5. This was mostly in specific cases that can be avoided, such as entering the model editor in gazebo while the TurtleBot3 model was active. Thus, crashes should happen less regularly than students had in the previous assignment. Additionally, if crashes do happen, there is enough reference material to solve problems, as all the tools are open source. So while the new tools might not necessarily solve all problems, they are a step in the right direction.

7.3 Q3

Q3 was focused on the technical details of what is available within ROS 2, the developed interface and mainly the SCT interface. The question was: what is a suitable level of abstraction for education using TurtleBot3? In the SCT interface the level of abstraction contains the min, max and mean of all laser data, a range of front facing laser and a range of lasers pointed at the wall, as well as the roll, pitch and yaw. Lastly, the speed and rotation can be controlled. This level of abstraction gives a clear goal for what students should use and does not overcomplicate the assignment. Because the statecharts can be made in a similar manner as with the EV3. Additionally the new tools grant access to the code where the data is abstracted. Thus, even if the students need a different level of abstraction, this can be created.

7.4 Additional Information

The TurtleBot3 brings additional improvements next to what was necessary to answer the questions. First of all, SCT has more features for designing statecharts, while State-

flow had more general modelling features because it is part of Simulink. These new features of SCT enable a more in depth use of statecharts. For example, a deep history allows for more complex systems than a shallow history. The second improvement is that everything is open source. As mentioned before, working with open source tools is inherently beneficial. Additionally, this also allows the students to continue working with the tools after the course has ended. Lastly, the experience with ROS 2 and an Eclipse based tool can be used in future work as these are used in industry and research.

7.5 The Main Question

To conclude, the new tools have positively completed all sub questions. One of the downsides is losing the experience with Stateflow and Simulink. However, SCT still gives useful experience, as Eclipse based tools are the second most used modelling tools in the embedded domain. Additionally, there are even more improvements next to what the sub questions required. So, the TurtleBot3 can be used as a replacement for the EV3 in an assignment to develop an application for an embedded system using statecharts. Thus, the TurtleBot3 Burger is a suitable platform to teach application programming for embedded systems using statecharts.

Chapter 8

Future work

To fully test if the TurtleBot3 can be used for education, it is important to let students use it. Thus to further quantify the results of this report, student feedback will need to be gathered on the redesigned assignment. Additionally by using the new tools in education, a more precise level of abstraction can be determined by what students need. Furthermore, another possibility for future work is testing the TurtleBot3 for other educational purposes. The TurtleBot3 main area of interest is autonomous robot navigation and it might be able to be used to educate students in applications, such as SLAM.

Bibliography

[1] Robin Amsters and Peter Slaets. Turtlebot 3 as a robotics education platform. In *International Conference on Robotics and Education RiE 2017*, pages 170–181. Springer, 2019.

[2] David Carrington and S-K Kim. Teaching software design with open source software. In *33rd Annual Frontiers in Education, 2003. FIE 2003.*, volume 3, pages S1C–9. IEEE, 2003.

[3] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[4] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling*, 17(1):91–113, 2018.

[5] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. In *Proceedings of the 13th International Conference on Embedded Software*, pages 1–10, 2016.

[6] Swas Oajsalee, Suradet Tantrairatn, and Sorada Khaengkarn. Study of ros based localization and mapping for closed area survey. In *2019 IEEE 5th International Conference on Mechatronics System and Robots (ICMSR)*, pages 24–28. IEEE, 2019.

[7] Theodore Pachidis, Eleni Vrochidou, Cristina Papadopoulou, Vassilis Kaburlasos, Snezhana Kostova, Mirjana Bonković, and Vladan Papić. Integrating robotics in education and vice versa; shifting from blackboard to keyboard. *International Journal of Mechanics and Control*, 20(1):53–69, 2019.

[8] Sasha Pietrzik and Balasubramaniyan Chandrasekaran. Testing autonomous path planning algorithms and setup for robotic vehicle navigation. In *2018 9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pages 485–488. IEEE, 2018.

[9] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[10] Louis van Zutphen, supervisors Benny Akesson, and Edwin Steffens. Gazebo simulation fidelity for the turtlebot3 burger. *Bachelor thesis UvA*, 2020.

Appendix A

Zoom in on the final Statecharts

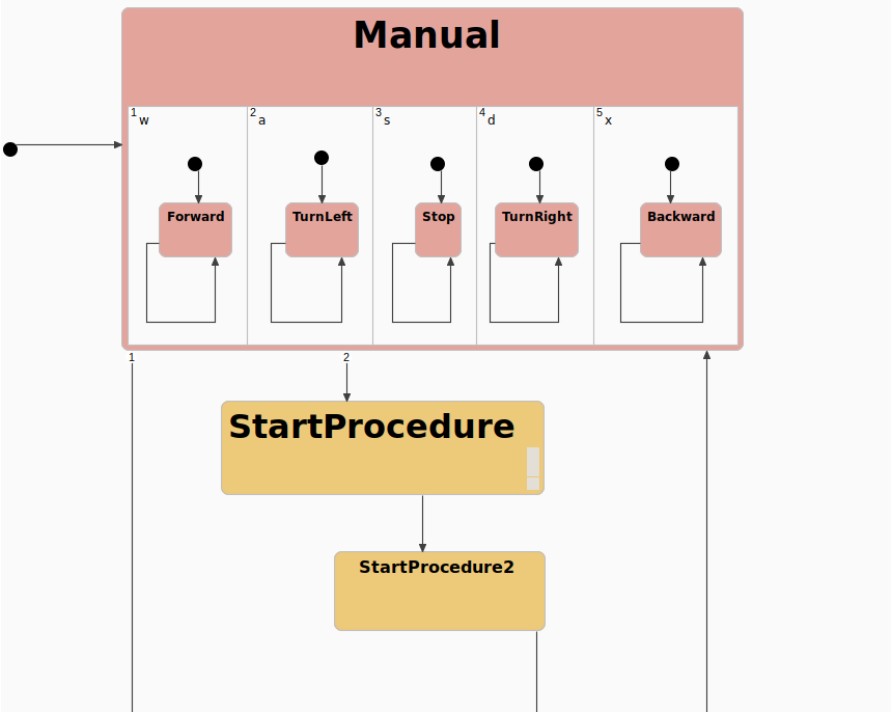


Figure A.1: The manual component of the statecharts, as well as the second step

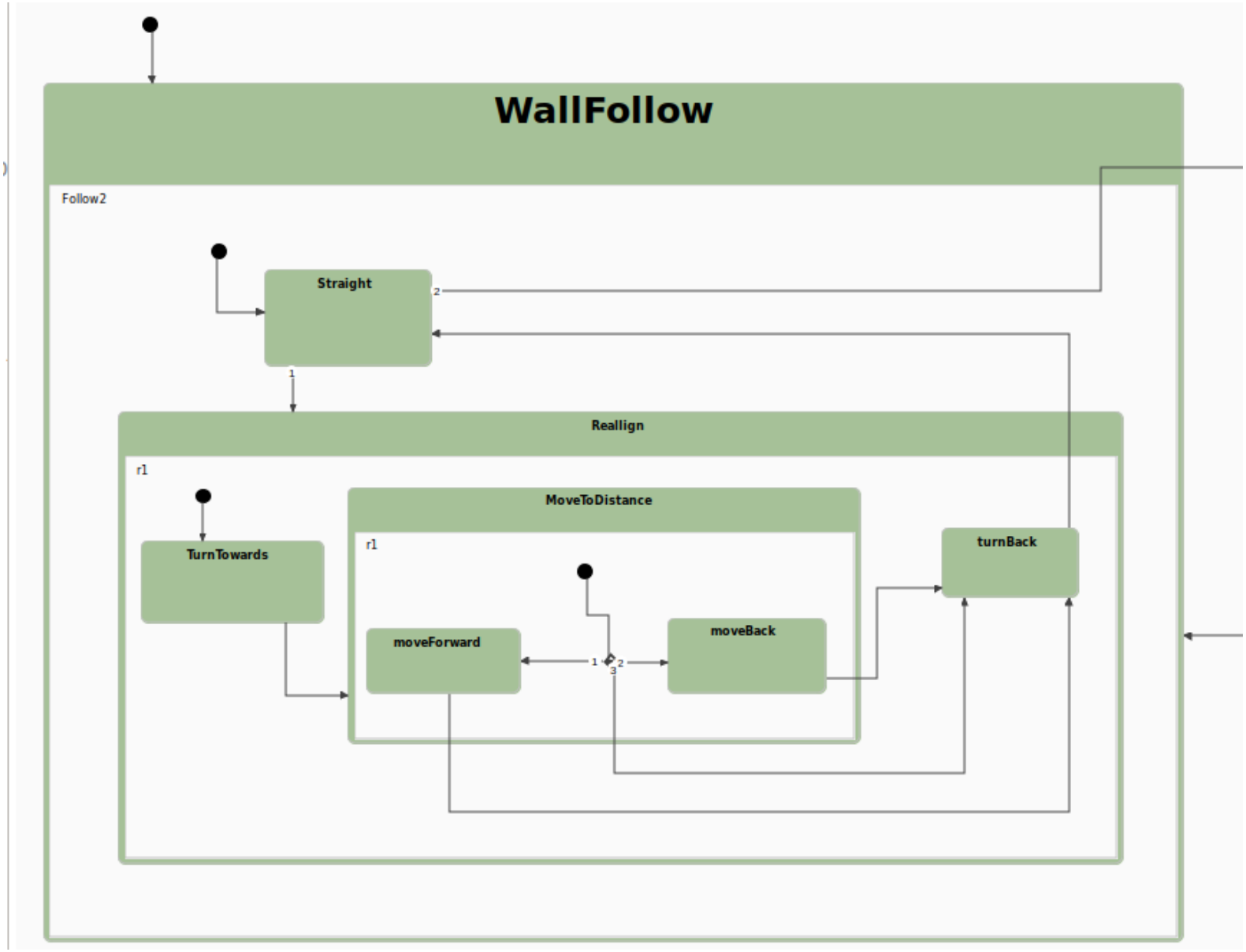


Figure A.2: The wall following component of the statecharts

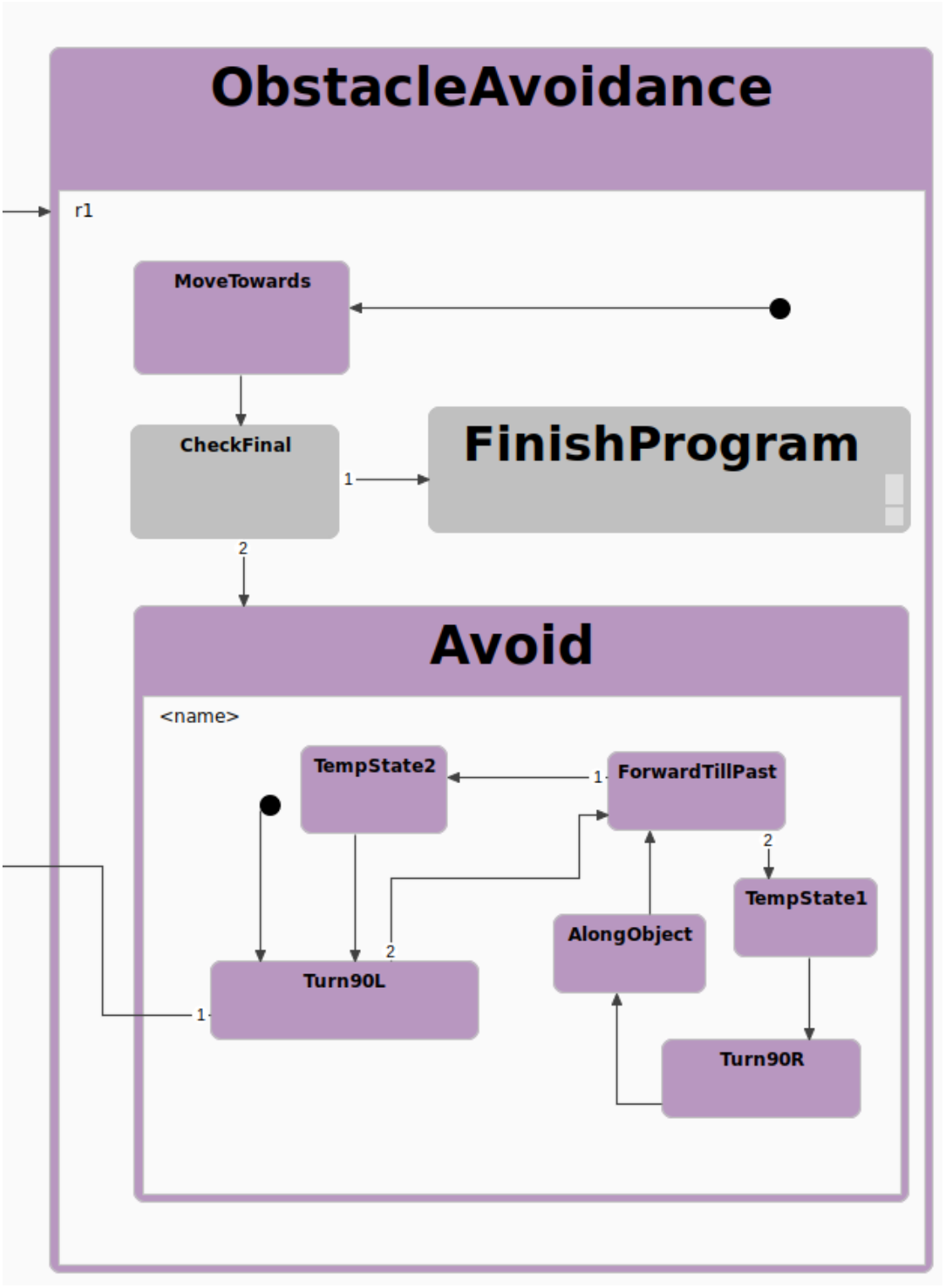


Figure A.3: The object avoidance and parking components of the statecharts

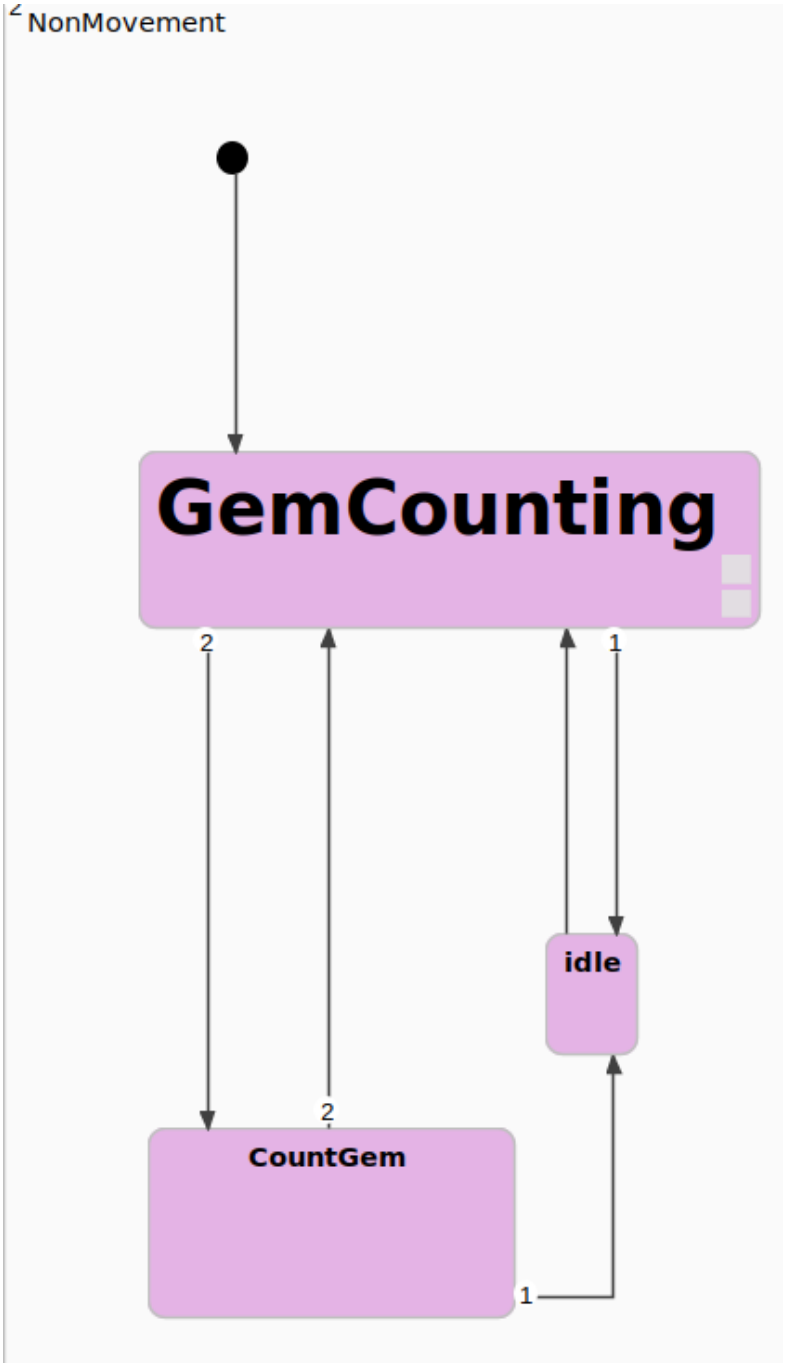


Figure A.4: The gem counting component of the statecharts