NTNU

Norwegian University of
Science and Technology

Scalable Self-Adaptation Control system for simulated transport robots

Magnus Karste Oplenskedal

Master of Telematics - Communication Networks and Networked Services
Submission date: July 2016
Supervisor: Peter Herrmann, ITEM

Norwegian University of Science and Technology
Department of Telematics

Scalable Self-Adaptation Control system for simulated transport robot

Author:
Magnus Oplenskedal

Supervisor:
Peter Herrmann

Master Thesis

Department of Telematics
Norwegian University of Science and Technology

Spring 2016

Problem Description

The goal of this master thesis was to design control software for simulated transport robots using model-based engineering. As a proof of concept there will be developed a prototype of the control software using the OSGi framework to accommodate a highly scalable, modifiable system, where parts of the system can be installed, started, stopped, updated and uninstalled without requiring a recompilation of the whole system.

The system should be designed in a way that promotes an easy porting from simulation to a real robot. To assist the development of the highly modularized real-time prototype, the following tools will be used; Eclipse plug-in Reactive Blocks (2) and the OSGi framework(3). The robot DiddyBorg(4) will be used as a model for the simulated robot.

Abstract

This master project aimed to explore, research, develop and evaluate the design and creation of control software for simulated autonomous transport robots. A proof of concept were to be created using the Reactive Blocks tool and the OSGi framework. Through literature study and development, a highly modularized system design was created, fit to tackle the problem domain. It allowed for the development of a modifiable control system able to move a simulated robot from a position to another using simulated robot components. The prototype renders a simulated robot on the screen, where it can be observed moving to a destination specified by the user of the system. The two technologies, Reactive Blocks and OSGi, were merged successfully to handle the complexity of a real-time robotics system consisting of several concurrent, independent sub-systems.

Preface

This report describes the work and research done, as well as the methodology and results achieved for the Master Thesis "Scalable Self-Adaptation Control system for simulated transport robot", held at Norwegian University of Science and Technology spring semester 2016.

I would like to thank my supervisor, Peter Hermann, for his continued advice, support and counseling throughout this project.

Contents

List of Figures

Part I

Introduction and Methodology

Chapter 1

Introduction

In safety critical domains like aviation, railroading, automotive, and robotics, one uses autonomous cyber-physical systems that interact with each other in the same physical space. Systems controlling robots, avoiding collisions with each other and objects in the physical world, getting from one destination to another, with the use of sensor input. When creating and testing control systems, simulation of the robot can be useful to avoid physical damage to real components and potentially humans. A fully simulated system can also be used to analyze the effects of malfunctioning hardware, and to assist the development of safe control software.

Using a framework like OSGi promotes the development of highly modularised software, which can be handy for the development of control software for transport robots where individual parts of the system controls individual physical parts of the robot. This can give developers the option to easily update, change and add new software to the system both during compile- and run-time.

1.1 Personal motivation

Personally I am genuinely interested in software architecture and design and loved the opportunity to hone my skills on creating an advanced system design for a problem domain I had little to non experience in. I got my first taste of the Robotics last semester, when I had a similar project description for my specialisation project (1). I really enjoyed working within the problem domain and with the challenges it provided. Through a specialization course held at NTNU, I learned about the OSGi specification and found it very intriguing to work with this interesting technology throughout my master project.

1.2 Readers Guide

This section aims to give the reader an overview of the parts, and chapters in this report.

Part I - Introduction and Methodology

The motivation behind the project, and how it was conducted.

Chapter 1 Introduction - Provides the motivation behind the project and the readers guide.

Chapter 2 Method - This chapter aims describes the research questions used to guide the literature research and the methodologies used to answer the project description.

Part II - Theory

Part II contains the information achieved during the research and work done before developing the control software.

Chapter 3 State of the Art - Describes articles of relevance to this master project, and in which way they are useful.

Chapter 4 Robot - Presents the robot used as a model for the simulated robot.

Chapter 5 Research Questions - Presents the results found when answering the research questions.

Chapter 6 Technologies - Briefly describes the two main technologies used in this project

Chapter 7 Control Software - Presents the theory, calculations and designs found, created and used to design the system and develop the prototype.

Part III - Results

This part describes the results discovered and achieved during this master project.

Chapter 8 System design and implementation - Presents the final system design and implementation of the system.

Part IV - Summary

This part discusses the findings in this project, presents a conclusion and lists areas of potential further work.

Chapter 9 Discussion - Discusses the process done during development, the system design and the final implementation of the software.

Chapter 10 Conclusion - Presents an conclusion based on the discussion in Chapter 9 and the results in Part III.

Chapter 11 Further work - Lists and discusses potential areas of expansion and future work for the system.

Chapter 2

Method

This chapter aims to explain and describe the methodologies, techniques and approach used to answer the master thesis.

2.1 Predefined requirements

The device used, device computer and development tool was predefined by the supervising professor for the the specialization project(1) I conducted last semester. Since the master project reside within the same problem domain as the specialization project, the pre-set conditions still stand. In addition to the previous conditions, the java framework for development of modularized systems, OSGi, is added to the list.

- Device: Diddyborg (4)
- Device computer: Raspberry PI (5)
- System development tool: Reactive Blocks (2)
- Java framework: OSGi (3)

2.2 Research Questions

Before the system design and implementation was initiated, a set of research questions were defined to narrow the research needed for the practical work. The

research questions were created to guide the literature research needed to answer the master thesis.

RQ1: What kind of software architecture is available for a highly modularised system for controlling transport robots?

Before designing the system it is important to know what kind of architectures already exist and if they can support a highly modularised system using OSGi(3) and Reactive Blocks(2).

RQ2: Are there any proof of concept systems, or control software systems created for robots using the OSGi Framework?

RQ3: How much of the work conducted in my specialization project can be used in this master thesis?

Can the research conducted during my specializations project (1) be used in the master project? Can parts of the code be used?

2.3 Method

The project work was started by defining several Research Questions which was later answered by doing a literature study. Based on this work a set of Minimum Viable Product Iterations were defined to guide the design of the system and the implementation of a prototype.

2.3.1 Minimum Viable Product

When designing the system for this project the Minimum Viable Product technique was used(20). This technique is handy the time estimation of a project is difficult, and the project has a hard deadline. The technique aims to create a system containing core functionalities as early as possible, a viable product with the bare minimum of functionalities. When this has been done the systems stakeholders will review and analyze the system and if accepted the system will undergo a new iteration where new functionality is added.

Before starting the development the functional requirements of the first iteration was defined. In addition to this several future iterations of the system was planned to make sure that the overall system design was achieved during the development.

By using this approach one functional iteration of the system will lead to another functional iteration, and even though, some times, not all iterations of the system can be implemented, one always has a functional system to fall back on.

MVP Iteration 1: Inter-modular Communication

The goal of the system was to create a highly modular system, using the Reactive Blocks tool(2) and the OSGi framework(3). By using these technologies all modules can have their own life-cycles, where the modules can be updated, installed, uninstalled, stopped and started. A system built on these modules require loosely coupled inter-modular communication and is therefore the first core functionality that should be designed and implemented.

MVP Iteration 2: Simulated sensor modules

After the inter-modular communication is in place, the next step is to add simulated sensor modules. Without sensor modules the control software for an autonomous robot cannot function. The second functionality to be designed and implemented should be the simulated sensors.

MVP Iteration 3: Simulate robot movement

Simulating the robots movement, in other words updating its simulated physical properties based on simulated movement should be the next step in the prototype.

MVP Iteration 4: Simulate robot moving to a destination

When moving the simulated robot is possible, a control algorithm for the prototype to an destination using its sensors, should be implemented.

MVP Iteration 5: Graphical simulation

Adding a GUI module for rendering the robots movement on the screen will help error-checking the control algorithm and provide users of the system a better way to observe the live simulation.

Part II

Theory

Chapter 3

State of the Art

The purpose of the master project was to design a scalable, modifiable control system for simulated robot and implement a prototype of the design. Before the design job was initiated it was important to research similar projects and systems, and learn from these. This chapter contains and presents a selection of the systems and articles and what could be learned from them.

3.1 Model-based engineering of Control Software for Simulated Robots

My specialization project autumn 2015 (1) was done as a part of my Master of Science degree conducted at NTNU. The goal of the project was to use model-based engineering to create control software for simulated robots, a goal similar to this master thesis.

The simulated robot should be able to move from position A to B using only simulated sensors and engines.

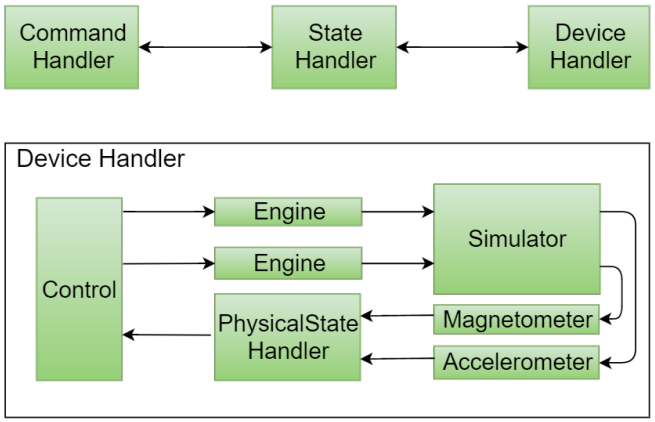

Figure 3.1: System design of the system created in my specialization project

The system is divided into separate modules, the modules are however contained in the same run-time application block and can therefore only be changed during development time.

In figure 3.1 we can see the 3 top-layer modules of the system. *CommandHandler*, *StateHandler* and the *DeviceHandler*. The CommandHandler module has the functionality to receive commands from external devices, these commands contain a destination for the robot. This destination is extracted from the command and sent to the StateHandler module. This module contains a queue of destinations, and if idle will start processing the first destination in the queue. The process is done by sending the current destination to the DeviceHandler module. This module controls the simulated physical parts of the robot, and can also be seen in Figure 3.1. The DeviceHandler sends the destination to its inner *Control* block .

The *Control* module contains the control algorithm of the robot and based on the current physical properties of the robot, the current position and the current destination, sets the power output of the engines to steer the robot towards its destination.

The *Engine* modules is just placeholder modules in this system, and sends the engine output to the simulator module.

The *Simulator* module uses the current physical properties of the robot to simulate the effect the engines power would have on real sensors, and sends this simulated sensor output to the sensor modules Magnetometer and Accelerometer.

The *Magnetometer* and *Accelerometer* modules also act as placeholder modules and pass the data on to the PhysicalState Handler module.

The *PhysicalState Handler* module uses the sensor output together with the robots current physical properties to calculate the new updated physical state of the robot.

The updated physical state of the robot is in turn sent to the Control module and used to calculate the needed engine output to travel towards the robots destination.

3.1.1 Usefulness

The idea of creating separate modules for each of the physical parts is good, to keep the simulated robot as similar to the real robot as possible. It also makes the job of porting the system from the simulation to a real robot easier, since the modules are separated in a natural way.

The control algorithm contained in the Control module can be used by robots using the magnetometer and accelerometer sensors.

The idea of placing all the simulation specific code in its own separate module is good, this also helps porting the simulation to a real robot.

3.2 Concurrent Planning and Execution for Autonomous Robots

The article written by Reid Simmons (7) describes the use of TCA (Task Control Architecture) in a system that walks a legged robot through rugged terrain. The walking system was originally implemented in a sense-plan-act sequential cycle, but was modified to concurrently plan and execute steps.

The interesting part of this article was Simmons use of the Task Control Architecture, because of this a short description of the architecture follows:

The *Task Control Architecture* provides a general framework for controlling distributed robot systems(7). In particular TCA supports distributed processing, hierarchical task decomposition, temporal synchronization of sub tasks, execution management, resource management and exception handling. The architecture is built up by a number of task-specific modules and a central module, the modules in turn communicate with each other by passing messages to the control module which routes the message to the appropriate module.

The TCA uses several different types of messages, Query messages are used for one module to get information from another and are blocking while awaiting a reply.

Non-blocking commands such as goal, command and monitor messages are used to create hierarchical plans. To coordinate messages the TCA uses: resources, hierarchical task trees, and temporal constraints. A resources is a set of message handling procedures and a capacity, by default all messages handled by a module are grouped as a resource of unit capacity. By doing this the TCA can only send one message to a module at the time, while queueing up additional messages. Resources can be reserved by modules, giving the module exclusive access. This can be used to synchronize resources. TCA records the sender and receiver of each message in a dynamically created hierarchical task tree. In these trees a sequential-achievement constraint can be set between two nodes. In this way all messages under the first node must be completed before the second nodes messages are handled.

3.2.1 Usefulness

The Task Control Architectures message-routing is an interesting approach to cooperation between modules of the system and might be useful for the design of my system. Reid Simmons(7) explains how they used the TCA to implement concurrency between the planning and acting phase of the robots movement. This can be relevant in my system, but for later iterations of the design.

3.3 An Architecture for Sensor Fusion in a Mobile Robot

The paper written by Steven A. Shafer, Anthony Stentz and Charles E. Thorpe describes sensor fusion in the context of an autonomous mobile robot(8). They describe the software architecture of a system they are building called the NAVLAB. The system is created for a vehicle, named the NAVLAB vehicle, a commercial truck modified with sensors, electronic controls and on-board computers and power generators. It is completely self-contained. The control software consists of computer controlled hydraulic drive system, controlled steering wheel, and processors to monitor and control engine functions

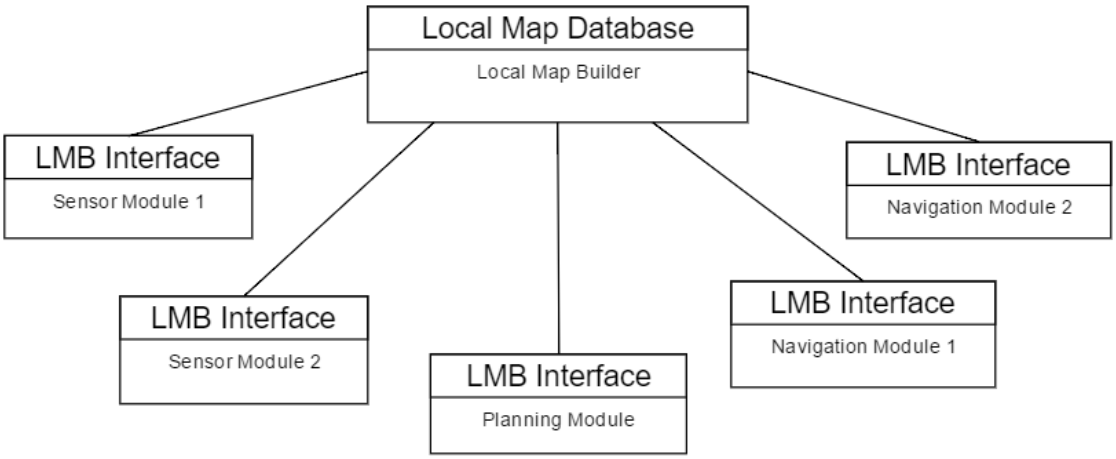

Figure 3.2: Process Structure of NAVLAB system

The system consists of several large modules, which are independently running programs. For example "map navigation and "road-edge finding by stereo vision". The modules are communicating together through a system Steven A. Shafer, Anthony Stentz and Charles E. Thorpe named CODGER(COmmunications Database with GEometric Reasoning). The program organization of the NAVLAB system is shown in Figure 3.2 where each of the boxes represent a separately running program. The Local Map Database is a central database all modules store and receive information from through a set of subroutines called LMB Interface (8).

The system structure has the characteristic of a blackboard system (9). The CODGER system of Steven A. Shafer, Anthony Stentz and Charles E. Thorpe differs from the blackboard system because their system consist of modules which are separate, continuously running programs an architecture they named *Whiteboard*. The communication between the modules consists of reading and writing data to the central database.

In the CODGER database, data is represented as tokens consisting of attribute-value pairs (8). Tokens can be scalars, arrays or geometric locations. Modules stores tokens in the database by a subroutine which sends the data to the central database. When a module needs data it creates what they call a *specification*. The specification can look like this example taken from (8):

tokens with ***type*** *equal to "intersection" and* ***traffic-control*** *equal to "stop-sign"*

In this example the database would return all tokens with type and traffic-control attributes satisfying the above constraints (8). With the use of this system the

NAVLAB provides asynchronous sensor fusion. An example taken from (8) explains how merging the results from module A(vision) and module B(rangefinder) into module C can occur in the following sequence:

1. Module A receives image at time 10 and writes results at time 15.
2. Module B receives data at time 12 and writes data at time 17.
3. At time 18, module C receives the result from module A and B. C uses vehicle coordinate system at time 12 for merging the data
4. Module C then requests module A's result, which was stored in VEHICLE time 10 coordinates to be transformed into VEHICLE time 12 coordinates. If necessary the system automatically interpolates coordinate transformation data. Module C can now merge A and B data since they are in the same time. At time 23, C stores the data in the database, with a coordination time of 12.

3.3.1 Usefulness

The problem domain is very similar to mine. The goal of my master project is to create a highly modularised system where the modules have their own life-cycles thanks to the OSGi framework. The way Steven A. Shafer, Anthony Stentz and Charles E. Thorpe uses a central Database for inter-module communication is very interesting, since it can effectively be done with OSGi as well. The use of a database to store values from the sensors can also be used to implement machine learning in later iterations of the system even though it is outside the scope of my project.

The way they created modules for each of the sensors and other modules for handling the data received from the sensors is also very interesting.

Their use of the *Whiteboard* pattern seems like a good idea, in this way the individual sensors are not restricted by a central controlling unit synchronizing their life-cycles. They can keep on measuring and storing data independently of the rest of the system.

3.4 Specifying Adaptations through a DSL with an Application to Mobile Robot Navigation

In (14) André C. Santos1,3, João M. P. Cardoso2, Pedro C. Diniz3 and Diogo R. Ferreira1 propose an adoptable system design for embedded systems based using a Domain Specific language. They aim to separate application and adaption concerns to allow for the reuse of adaption mechanisms.

3.4.1 Usefulness

Separating the adaption concerns to allow for reusable adaption mechanisms is a good idea, and supports the plan for my master project of creating a highly modularized system. Separating the adaption concerns into its own module will result in the possibility to update and exchange the adaption functionalities without to any modifications to the rest of the system. Since OSGi is a central part in my master project, and OSGi provides the possibility to update, stop, install, uninstall and start modules in run-time. A combination of their adaption strategy and the OSGi framework can give the module containing the adaption concerns the power to change the states of other modules in the system.

3.5 A Hardware-in-the-Loop Simulator for Distributed Robotics

Ritesh Lal and Robert Fitch (15) discuss self-reconfiguring modular robots, versatile robots that can adapt to their environment through changing its modules in run-time. They describe the difficulties of observing the full state of the system at any given time and propose an Hardware-in-the-loop simulator. They present a custom HIL simulator for distributed robotics which include small graphical displays to facilitate debugging.

3.5.1 Usefulness

Designing my system to have the possibility to be used in an Hardware-In-the-Loop setup is a great idea, and should be looked more into through the development of my system design and prototype.

3.6 Hardware in the Loop for Optical Flow Sensing in a Robotic Bee

Pierre-Emile Duhamel ,Judson Porter, Benjamin Finio, Geoffrey Barrows, David Brooks, Gu-Yeon Wei, and Robert Wood (16) describe their hardware in the loop system for simultaneous development and testing of different individual components and *RobotBees*.

3.6.1 Usefulness

Their idea of using a hardware-in-the-loop system which tests different components at the same time to find the best fitting components for their problem domain is a really good idea. This is something I can envision for the future of the system design in my master project. Testing sensors in the same system, running the same algorithms and then having the possibility to retrieve this data and compare it to select the best fitting sensor is a great idea for the future of my system design

Chapter 4

Robot

Control software for a robot can be either immensely complex or rather simple, it all depends on the robot that is controlled. A robot can be a human-like machine, a machine with arms and legs or it can be a car. A robot can be autonomous, semi-autonomous, or directly controlled by an external source. The goal of this chapter is to describe the robot simulated in my project, and the physical properties of the robot reflected in the simulation system.

4.1 The Diddyborg

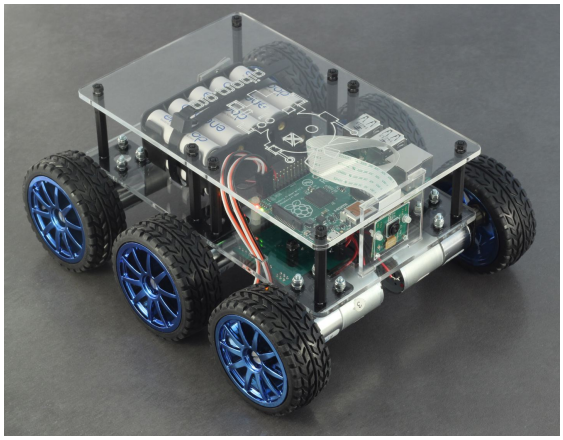

Figure 4.1: Image of the Diddyborg, the physical robot used as model for the simulation (21)

The robot used as a model for the simulated robot in the prototype is called a Diddyborg(4). The Diddyborg consists of a body, containing the Raspberry Pi and 6 motors. The motors are placed on the sides of the robot, 3 on each side. Each motor is connected to a wheel and all motors one one are connected to the same power output. The motors on the left, can be operated independently from the motors on the right and vice versa. The motors can be controlled by giving them a command with a value indicating amount of power the motor should use, this value is set from from 0-100%, in either relative forwards or reverse direction. To easier simulate the difference between power for the motor in the two different direction, a power range from -100 to 100 is proposed. See Figure 4.2 for overview over how the motor power is used to control the robot.

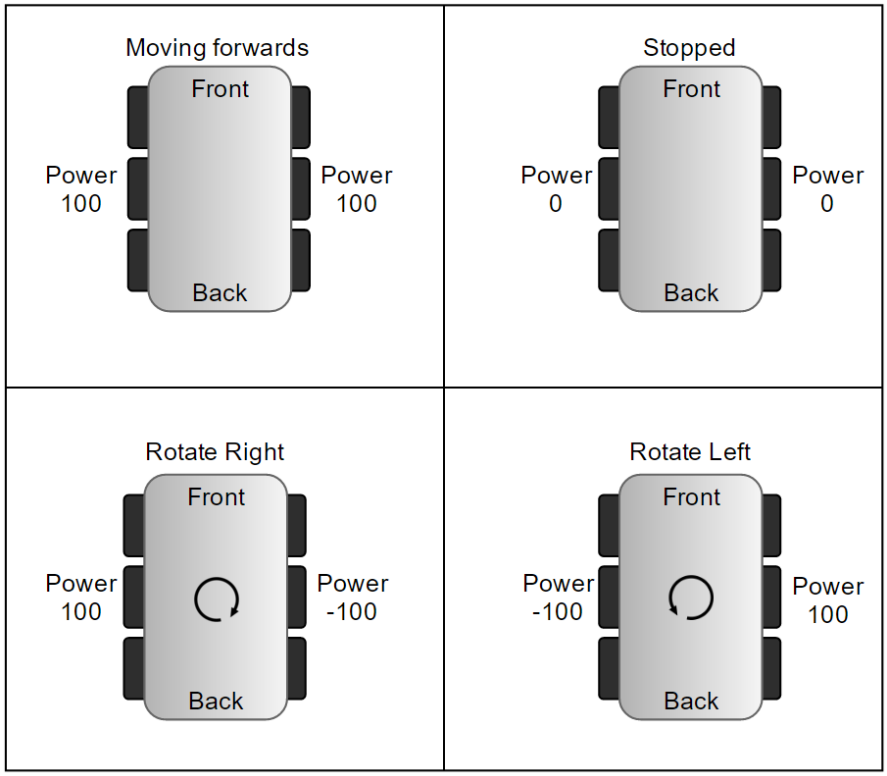

Figure 4.2: Model of Diddyborg seen from atop, and how powering the left and right motors is used to control the robot

There are no axles which can be be used to change the angle of the wheels. And as such rotating the robot can only be done by differentiation of the power output to the motors.

4.2 Sensors

The following sensors are proposed for the prototype of the system; *Magnetometer* and *Accelerometer*. This because the sensors are available for the robot Diddyborg, on the chip *XLoBorg* (22). An accelerometer works by measuring the acceleration in relative x, y and z directions. An magnetometer works like a compass and measures the earths magnetic-fields to find the current bearing of the device.

Chapter 5

Research Questions

This chapter presents and describe the answers found to the research questions defined in section 2.2. The questions were created to guide the research deemed necessary for the design of the system, and the implementation of its prototype.

RQ1: What kind of software architectures is available for a highly modularised system for controlling transport robots?

This questions was aimed towards state of the art architectures for the problem domain my master thesis resides in. Several architectures were researched and documented in chapter 3.

The *Task Control Architecture* discussed in section 3.2 describes an architecture using an interesting message routing system, where communication between the modules were routed through a central module. The article was mainly focusing on creating an architecture for concurrency in the planning and acting phase of the robots movement cycle.

Steven A. Shafer, Anthony Stentz and Charles E. Thorpes Whiteboard System NAVLAB discussed in section 3.3 is concerned with a problem domain very similar to my master thesis. They are creating a system consisting of multiple independent programs communicating through a central database. They use the same modularisation strategy of placing sensor software and navigation software in separate modules. Sensor modules write sensor data to the database and navigation modules read the data to be able to navigate the robot to its destination.

RQ2: Are there any proof of concept systems, or control software systems created for simulated mobile robots using the OSGi framework?

I could not find any other works done in the mobile robot simulation domain using the OSGi framework and Reactive Blocks. The NAVLAB system described by Steven A. Shafer, Anthony Stentz and Charles E. Thorpes however consists how highly decoupled modules, which might be applicable to a system built on the OSGi framework.

RQ3: How much of the work conducted in my specialization project can be used in this master thesis?

The problem domain of my specialisation project (1) have a lot in common with my master thesis, and can be seen as a stepping stone towards the problem description of my master thesis. The two projects differ in the focus of the system design. Where the specialisation project was mainly focused on the control software, creating software that could be used to automate a simulated robot, the master project is focused on the system architecture and design of the control software and the ability to build the system using the OSGi framework.

There are also similarities which can be used in the master thesis, for example the use of the same type of robot and sensors allows the use of the same control algorithms.

Parts of the code might be of use, for example all the POJO(Plain Old Java Object) classes, see Figure 5.1, can be used since these are the same for both projects.

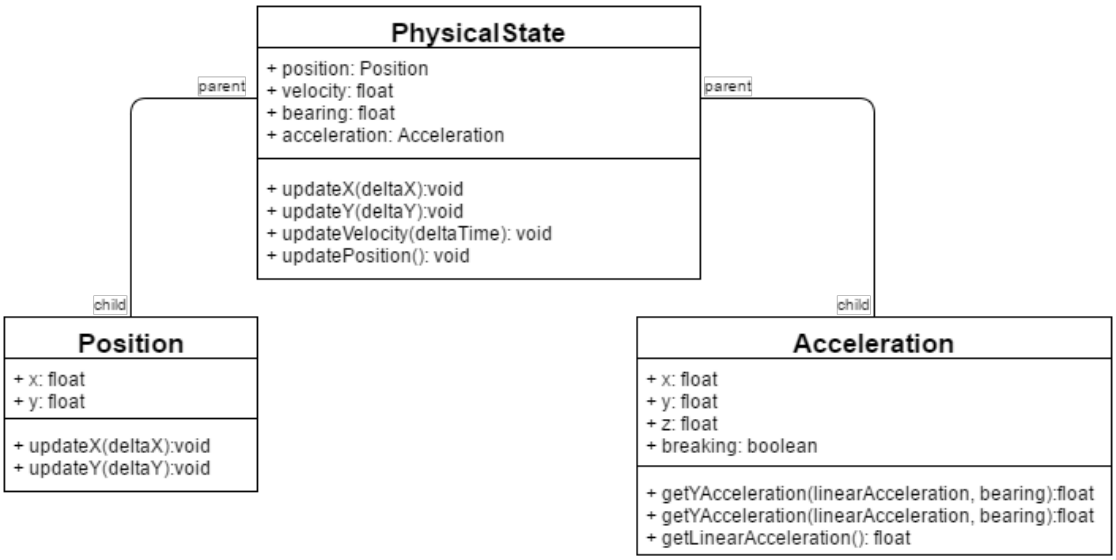

Figure 5.1: Class diagram showing relations between classes from my specialisation project (1)

The research questions gave me an idea of where to start when designing the control software, and the state of the art chapter gave me inspiration on what kind of architecture and design I could create.

Chapter 6

Technologies

This chapter aims to give the reader a short summary of the two most important technologies used in this project. The Reactive Blocks tool (2) and the OSGi framework (3).

6.1 Reactive Blocks

Reactive Blocks (2) is an Eclipse plug-in created for the development of reactive real-time software systems. A system in Reactive Blocks consists of building blocks which are subsystems or functionalities, and can be composed of each other. The blocks created in Reactive Blocks are stored in on-line libraries and can easily be reused. This is done by dragging and dropping building blocks into projects. Building blocks behaviours are modeled by UML activities that contain UML call actions. The interface of a block is created by defining an *External State Machine*(10). ESM are state machines defining when input and output flows are allowed to be passed through the pins at the Blocks edges(11).

Reactive blocks can analyse functional correctness since both activities and ESM are supplemented with formal semantics (13). Reactive Blocks projects are automatically transformed into Java (12).

6.2 OSGi

The OSGi specification allows for the development of modular systems in the Java programming language implementing an dynamic component model. The modules are called Bundles, JAR's with extra metadata allowing them to have their own separate life-cycles. Each bundle can be remotely installed, started, stopped, updated and uninstalled without requiring a full reboot and recompilation of the system. During run-time,a bundle can exist in one of the six following states:

- UNINSTALLED
- INSTALLED
- RESOLVED
- STARTING
- STOPPING
- ACTIVE

A bundle in the state UNINSTALLED is uninstalled from the framework and cannot be used. A INSTALLED bundle is installed in the framework, but not yet resolved. A RESOLVED bundle is a bundle which is installed, and has had all its dependencies checked such that it will not break the system in any way when it is started. STARTING and STOPPING are temporary states that the bundle enter when it is in the process of starting or stopping. The ACTIVE state is entered by the bundle when it has been resolved and is finished in the STARTING state. The bundle can now be actively used by the framework and other bundles in it.

All bundles exists and share the same *Bundle context* in the framework. It is through this bundle context that a bundle can access the services provided by the framework, other modules or use one of the following communication techniques:

The OSGi framework allows for bundles to share functionality with other bundles with the use of a service layer. In the service-layer bundles can communicate by using a publish-find-bind model for POJIs(Plain Old Java Interfaces) and POJOs(Plain Old Java Objects). A bundle can register a service in the framework, which other bundles can extract and use.

The framework comes with an built-in event service called the *Event Admin Service*. Using this service, bundles(modules) in the system can publish events with topics in the framework, and other bundles can subscribe to these topics to receive the events.

Chapter 7

Control Software

Chapter 7 contains the research made and the necessary formulas created to develop the simulated control software. The chapter is built up by first defining the properties of the simulated robot, and its predefined simplifications. After this the work and research done for each of the MVP iterations defined in section 2.3.1 will presented.

7.1 The Simulated Robot

Before the design of the MVP iterations was started, the properties of the simulated robot had to be defined. This to guide the functional and non-functional requirements of the software. The robot portrayed in chapter 4, will be the model for the simulated robot.

The Diddyborg (4) has 3 wheels connected to individual motors on each side of its body, these motors can however not be controlled separately. Therefore, the simulated robot has only 1 engine on each side of its body. To keep the control algorithm and system simple and to be able to easier focus the work on the underlaying infrastructure, only the Magnetometer and the Accelerometer sensors are simulated.

7.2 Simplifications

The control algorithm itself is not the most important part of this system, at least not in the first iteration of the system. To keep the control algorithm simple a

number of simplifications were set for the simulated robot.

- Movement: The simulated robot has 4 movement states, Forward, Rotate and Stopped. The robot can only do one of these at the time, and as such the robot is standing still while rotating, and cannot change direction while moving.
- Acceleration: The simulated robot only accelerates when moving forward or decelerates when breaking. And as such when rotating the robot will not change its position.
- Acceleration in x, y and z direction: The robot can only move in the xy plane, and as such the acceleration and movement in the z-direction is disregarded.

7.3 Minimum Viable Product Iterations

Before starting the work on the first Minimum Viable Product iteration, a rough sketch of the system design was needed. This was important since the first MVP iteration was how to deal with inter-modular communication. In the beginning this was just a rough division of responsibility into modules with high cohesion, as can be seen in figure 7.1

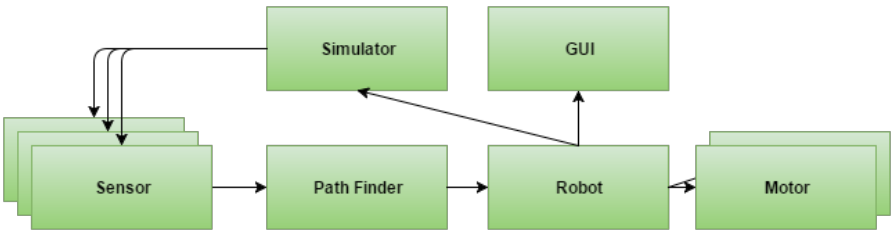

Figure 7.1: Early sketch of system design

7.3.1 MVP Iteration 1: Inter-modular Communication

From the work done during the literature research, and technology research shown in chapter 6 a number of possible communication technologies and ideas were found.

The first step was to expand system design, to map what kind of data one module needs and what it can provide for others. Figure 7.2 shows an updated version of the early system design described in Figure 7.1.

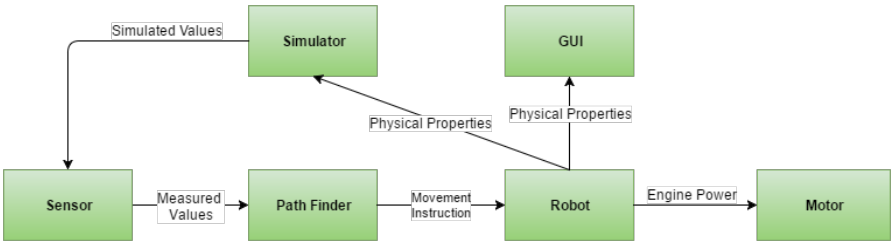

Figure 7.2: Updated early sketch of system design with defined communication parameters

In Figure 7.2 we can see an overview of the modules as well as the data sent between the modules. This is a good starting point for the creation of the inter-modular communication. The first data needed in the system is the sensor output, and as such the first interface designed was the interface between the *Sensor* and the *Path Finder* modules.

Sensor data registration

The communication between the sensor and path finder modules are one-way, from the sensor to the path finder. The data sent was postulated to consist mainly of floating values, for example the bearing of an device registered with a Magnetometer, or the three floating numbers registered by an Accelerometer. In other words the data sent from the sensor module will be one or more floating values pr. measurement.

The Path Finder module needs to be able to accept data from several Sensors at the same time. And to get the newest registered value from the sensors, even though they do not register data at the same time or register data at the same frequency. For example Sensor A registers data at time 5, 10, 15 and so on. It registers data at an interval of 5 seconds. Sensor B registers data at time 2.5, 5, 7.5, 10 etc, in other words it registers data at another frequency than sensor A. The Path Finder Module should always get the newest measurement from all sensors. The problem is how to synchronize the data received from the sensors. At what time should the Path Finder run a new calculation on the registered measurements? How can the Path finder store the measurements from the different sensors while waiting for data from other slower sensors? How can this sensors fusion be handled?

The DataAccess Service

The design used by Steven A. Shafer, Anthony Stentz and Charles E. Thorpe in (8), where all their sensor modules are independently running programs registering data in a central database came to mind as a good way to solve the sensor fusion problem. If all of the sensors write data to a database, the Path Finder module can extract data from the database at intervals and use this data to calculate the path of the robot. When the data from the sensors are registered in a database, the measurements might be also be used in machine learning, in later iterations of the software. This is however outside the scope of this master thesis.

The next issue is deciding which module should have the responsibility of the database. To keep high cohesion within the modules, and low coupling between them, a new module was proposed. A module with the sole responsibility of connecting to and communicating with the database. In Figure 7.3 a new iteration of the system design can be seen, with the addition of the *DataAccess* module.

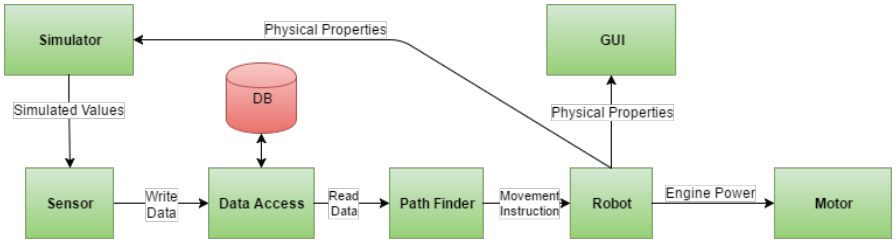

Figure 7.3: DataAccess module included in early system design sketch

When a sensor module wants to store data in the database, it has to do this through the Data Access module, the Path Finder must use the Data Access module to read data from the database. The Data Access module will thus work as a *Service* the Path Finder and sensor modules can use to read and write to the database.

Keeping low coupling between modules and cohesion between them is important when creating highly modular systems. The modules should have as little knowledge as possible about each other, while still being able to communicate and work together. How can this be achieved?

Publish-Subscribe Pattern

The *Publish-Subscribe* pattern (17) is a pattern where publishers publish events under a *Topic* without having any information about potential subscribers and if there are any subscribers at all. The subscribers subscribe to a Topic, without

having any knowledge of when events will arrive or where they come from, as long as they are published under the subscribed Topic.

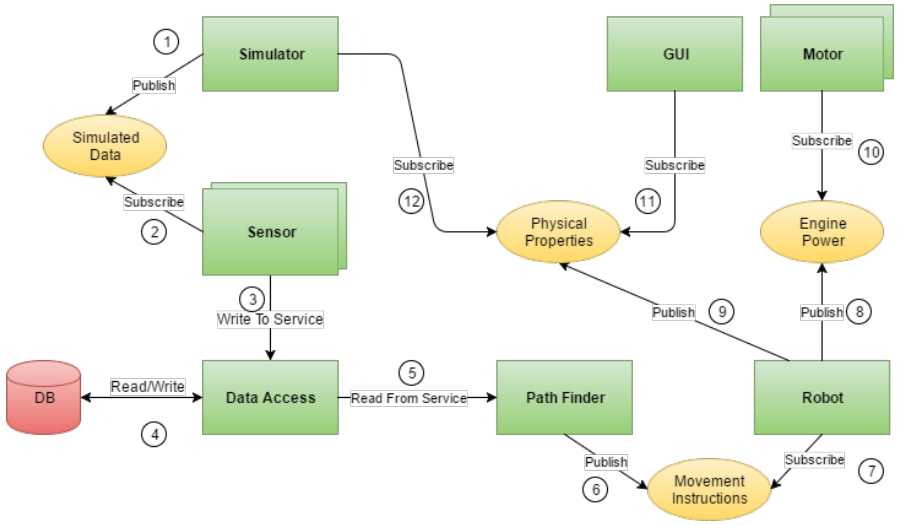

Figure 7.4: Inter-modular communication diagram, showing the type of communication used between the modules in the system design.

Figure 7.4 shows the different modules in the system design (green squares) and topics they publish/subscribe to (yellow ellipses). The communication between the modules run in a loop, depicted by the steps in the figure, where step *1* is the simulator module publishing simulated data, step *2* is the sensor modules subscribing to the simulated data topic. Step *3* is the Sensor modules using the Data Access module to write data in the database. The next step *4* is for the Data Access to store the data in the database, which the Path finder reads in step *5*. In step *6* the Path Finder module Publishes its data, and in step *7* the robot module subscribes to this data. The Robot module publishes two topics, the Engine Power topic in step *8* which the Motor modules subscribe to in step *10* and the Physical Properties topic published in step *9*. Both the GUI module and the Simulator module subscribe to this topic in step *11* and *12*, this is an example of how to modules can subscribe to and use the same data in parallel, without having any knowledge of each other and the Robot module having no knowledge of either of its subscribers.

The Simulation Loop

The internal modular communication shown in figure 7.4 is in this project called *the simulation loop* and works as a feedback loop where the data measured by the

sensors are used to control the simulated robots engines, which in turn is fed back to the simulator module that simulates the data input to the sensors, Figure 7.5

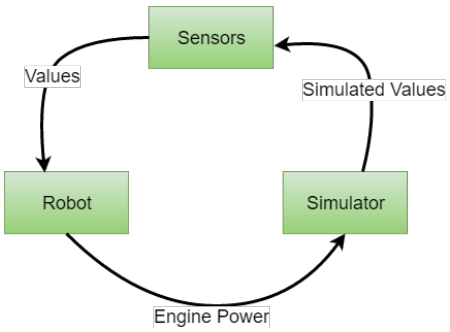

Figure 7.5: The Simulation Loop

7.3.2 MVP Iteration 2: Simulated sensor Modules

With the inter-modular communication in place, the next step will be to implement functionality for the sensor modules to register data in the database, and doing this through the Data Access module.

SQLite Database

The Data Access module has the responsibility of connecting to, writing and reading from the database. Using a SQLite database is proposed because this allows for a self-contained database system where the Data Access module can have the responsibility of creating the database file itself. This way the system will not have to rely on external software for database creation and control.

"SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine." (18)

SQLite databases does not have a separate server process, but reads and writes directly to ordinary disk files. When it comes to concurrency, SQLite databases allow for concurrent reading but not writing. This might be an issue for the control software where several Sensors are measuring and writing data in parallel. This issue can however be solved by using a buffer where sensors registers data, and then allowing the Data Access module to pull data from the buffer and store it in the database at certain intervals. After each pull the Path Finder can be allowed a read period where it requests the newly stored data in the database, see Figure 7.6.

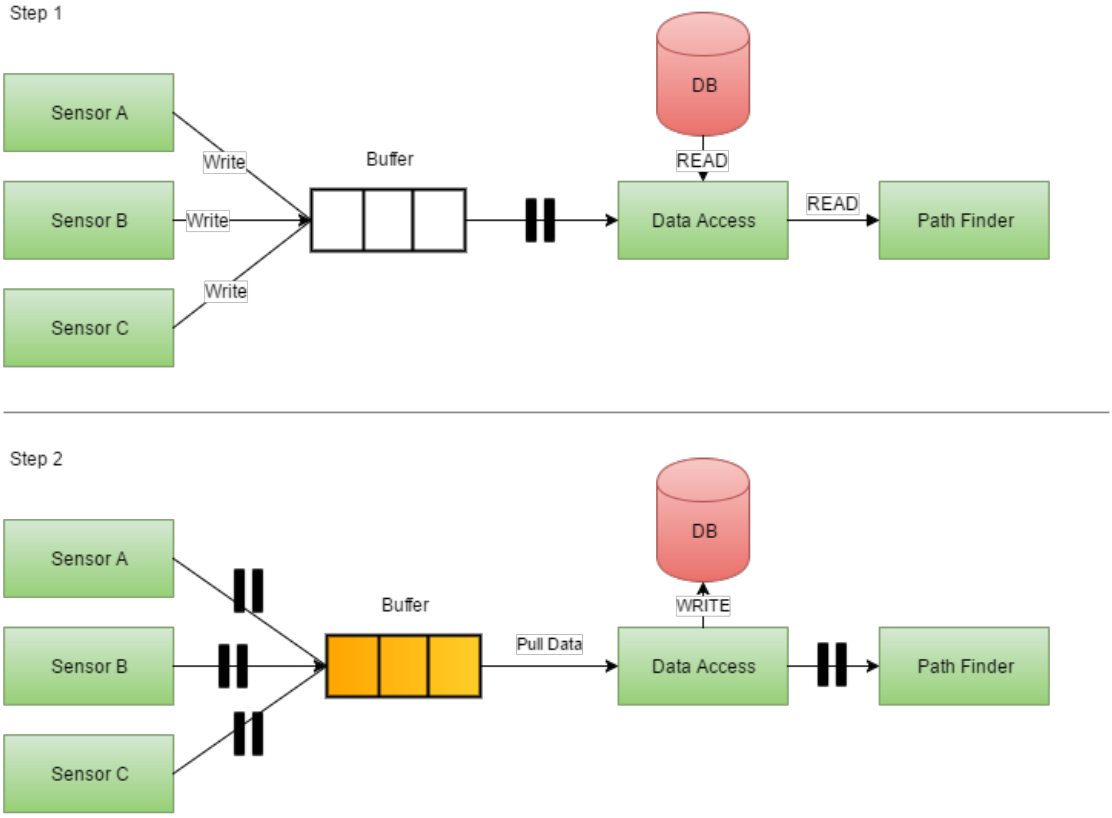

Figure 7.6: Step 1: Sensors registering data in a buffer at the same time as Path Finder reads data from database. Step 2: Data Access pulls data from buffer and stores in the database

With this functionality in place the Sensor modules can register data in a buffer concurrently, which the Data Access module pulls from the buffer and stores in the database. The sensor modules should be simple containing the functionality to subscribe to data sent from the Simulator module and store this data in the database.

7.3.3 MVP Iteration 3: Simulate robot movement

To be able to simulate a moving robot, one first has to calculate the robots simulated physical properties. The physical properties that will be included in the prototype are the following:

- Position

- Bearing
- Acceleration
- Velocity

Relative Position

Changing the robots position over time is what movement is actually all about. Having positional awareness is therefore the first property that is needed. With the sensors defined for the simulated robot, only the relative position can be found, e.g. the position of the robot in relation to where it started. Moving the simulated robot will work by iterating the relative position of the robot based on the calculations using the following physical properties; bearing, acceleration and velocity.

Bearing

To be able to calculate the relative position of the simulated robot, the bearing and linear acceleration is needed. The bearing is the direction the robot is heading and is measured by a magnetometer. The data output by a magnetometer is usually a value between 0 and 360 degrees. The relationship between the four cardinal directions and the output from the simulated magnetometer for this prototype can be seen in figure 7.7

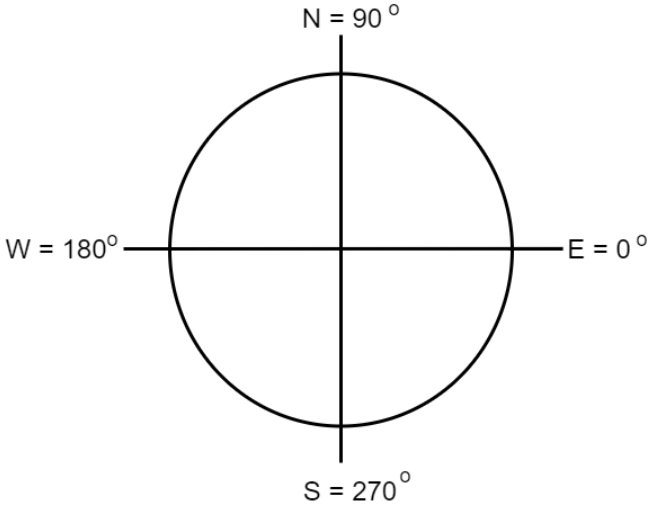

Figure 7.7: Cardinal to bearing relation for simulated magnetometer

Simulating the bearing of the robot and the change of it is done by updating the current bearing of the robot for each cycle of the *Simulation Loop* with a predefined "degrees pr. time unit" when the robot is in the rotating state.

In addition to the bearing, the linear acceleration of the robot is needed to calculate the relative position.

Acceleration

The accelerometer measures acceleration in the relative x, y and z directions, relative in the sense that its relative to the x, y and z direction of the device.

The simulated accelerometer is placed directly on top of the simulated robot, and as such the x direction of the accelerometer is parallel to the forwards direction of the robot, and the y direction is perpendicular to the forwards direction. This will result in an positive x acceleration when the robot is moving forwards and an negative acceleration when the robot is breaking. Acceleration in z-direction will be neglected in the prototype of the system, it will be assumed that the simulated robot is moving in a the xy-plane only. The acceleration of the robot will, in this iteration of the prototype be the same as the acceleration measured in x-direction since the robot can only be moving forwards or rotating, it can not turn while moving forwards creating acceleration in both x and y directions (see *Simplifications*). In later iterations of the prototype, it can however be interesting to have the possibility to measure linear acceleration from both x and y directions. The linear acceleration can be found by using the *Pythagorean theorem*, see Equation 7.1. Where the hypotenuse is the linear acceleration and the x and y accelerations are the two other sides of the triangle.

$$La = \sqrt{Xa^2 + Ya^2} \tag{7.1}$$

Where La is the linear acceleration of the robot, and Xa and Ya are the acceleration in x and y directions. With the linear acceleration of the device at hand, the velocity of the device can be found.

Velocity

The delta velocity Δv of the robot can be found by using the linear acceleration La of the device, and time Δt since last update, see Equation 7.5. Δv is the speed

increased by accelerating with the acceleration La over the time Δt.

$$\begin{aligned} &Acceleration = m/s^2 \\ &\Delta Time = s \\ &\Delta Velocity = Acceleration * \Delta time = m/s^2 * s \\ &\Delta Velocity = m/s \end{aligned} \tag{7.2}$$

With the Δv of the robot at hand, the current velocity v can be found with the following equation:

$$v = v + \Delta v \tag{7.3}$$

And the velocity Xv in x direction and Yv in y direction can be found with the following trigonometric functions, where b is the current bearing of the device:

$$\begin{aligned} Xv &= v * cos(b) \\ Yv &= v * sin(b) \end{aligned} \tag{7.4}$$

Updating relative position

With access to the speed of the robot in both x and y directions, we can finally update the relative position of the robot with the following Equation:

$$\begin{aligned} x &= x + Xv * \Delta Time \\ y &= y + Yv * \Delta Time \end{aligned} \tag{7.5}$$

Here we add the current speed in x and y directions, Xv and Yv, multiplied by the time since last update ΔTime to the current position of the robot x and y.

Movement

By using all the equations listed in this section we can update the relative position of the robot over time, simulating movement.

7.3.4 MVP Iteration 4: Simulate the robot moving to a destination

With the ability to simulate robot movement, the next step is to move it somewhere. In the prototype of the system the goal will be to move the robot to a destination using its sensors for navigation. The algorithm controlling the movement will reside in the *Path Finder module*, see Figure 7.4. In Figure 7.8 an activity diagram, showing the general flow of the control algorithm is depicted, this algorithm will be run by the Path Finder module for each iteration of the *Simulation Loop*.

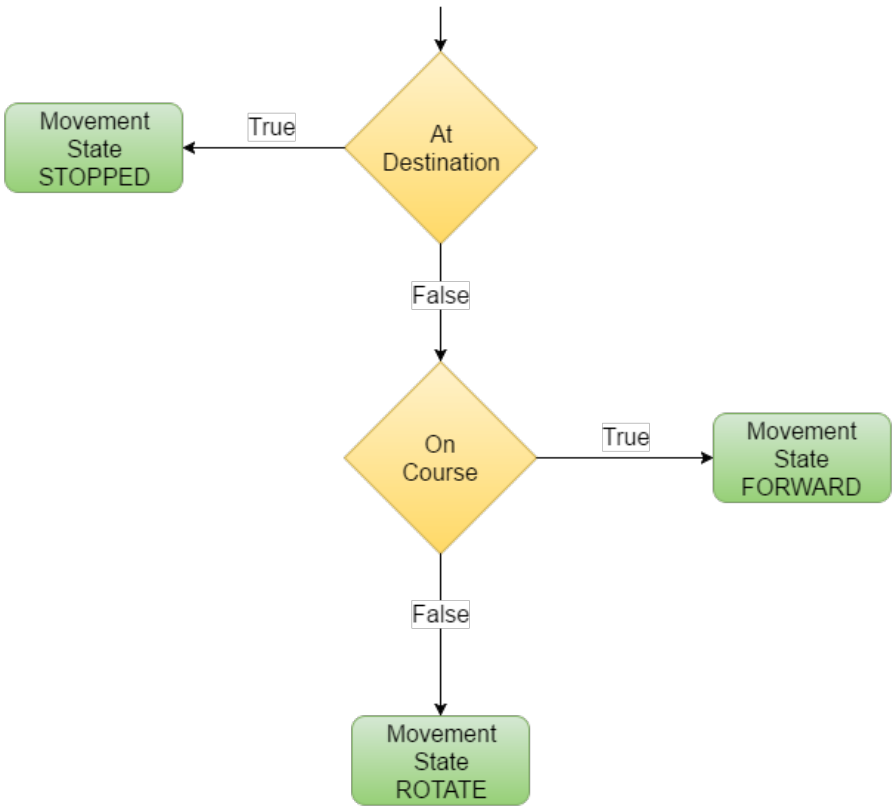

Figure 7.8: Activity diagram showing the Path Finders control algorithm

At Destination

In Figure 7.8 the first check the algorithm performs is to see if the robot is at its destination, if it is it should stop and if not the algorithm moves to the next step.

To check whether the robot is at its destination Equation **??** can be used to find the difference between the robots position and the destination.

$$\begin{aligned} \Delta x &= |x - destX| \\ \Delta y &= |y - destY| \end{aligned} \tag{7.6}$$

Comparing Δx and Δy with a predefined value for how much the x and y position of the robot is allowed to deviate from the actual destination, can be used to check if the robot has reached its destination, see Figure 7.9.

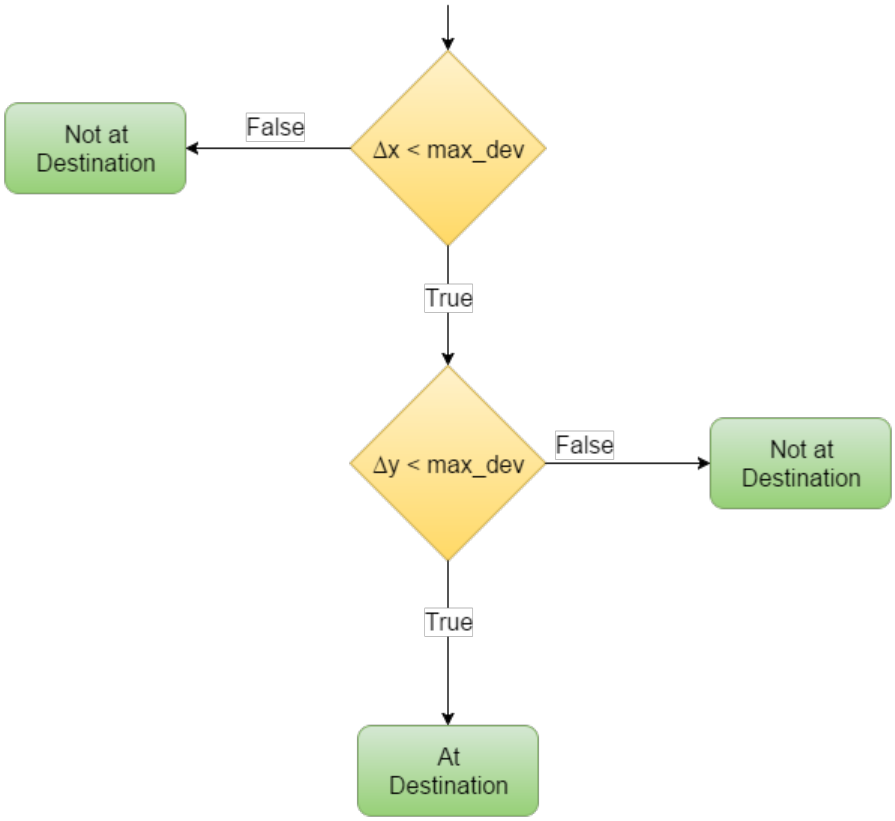

Figure 7.9: Activity diagram depicting the logic the robot uses to decide if it has reached its destination or not

On Course

If the robot has not reached its destination, the next step is to decide whether it should enter the Rotate or the Forward state, see Figure 7.8. To do this it has to

know if it is on course or not. The direction the robot needs to drive in to reach the destination is in this master project called the *Optimal Bearing*.

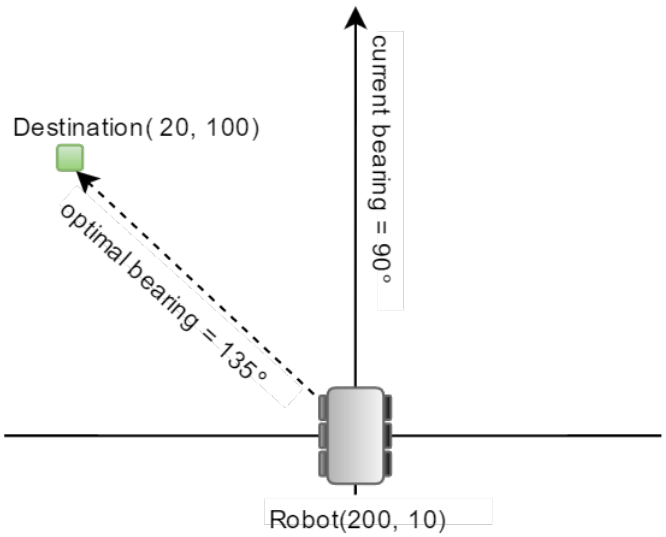

Figure 7.10: Optimal bearing for the robot to reach its destination

To calculate the optimal bearing the angle (α) between the robot and the destination must be found. This can be done by the following Equation:

$$\alpha = |\frac{arcTan(\frac{\Delta y}{\Delta x})}{\pi}| \tag{7.7}$$

Where Δx and Δy is found with Equation 7.6.

With α in hand we know the angle between the robot and the destination, however we do not know where in relation to the robot the destination exist. If its to the west, east, south or north of the robot. To find this we can compare the x and y coordinates of the position with the x and y coordinates of the destination, following the logic depicted in Figure 7.11

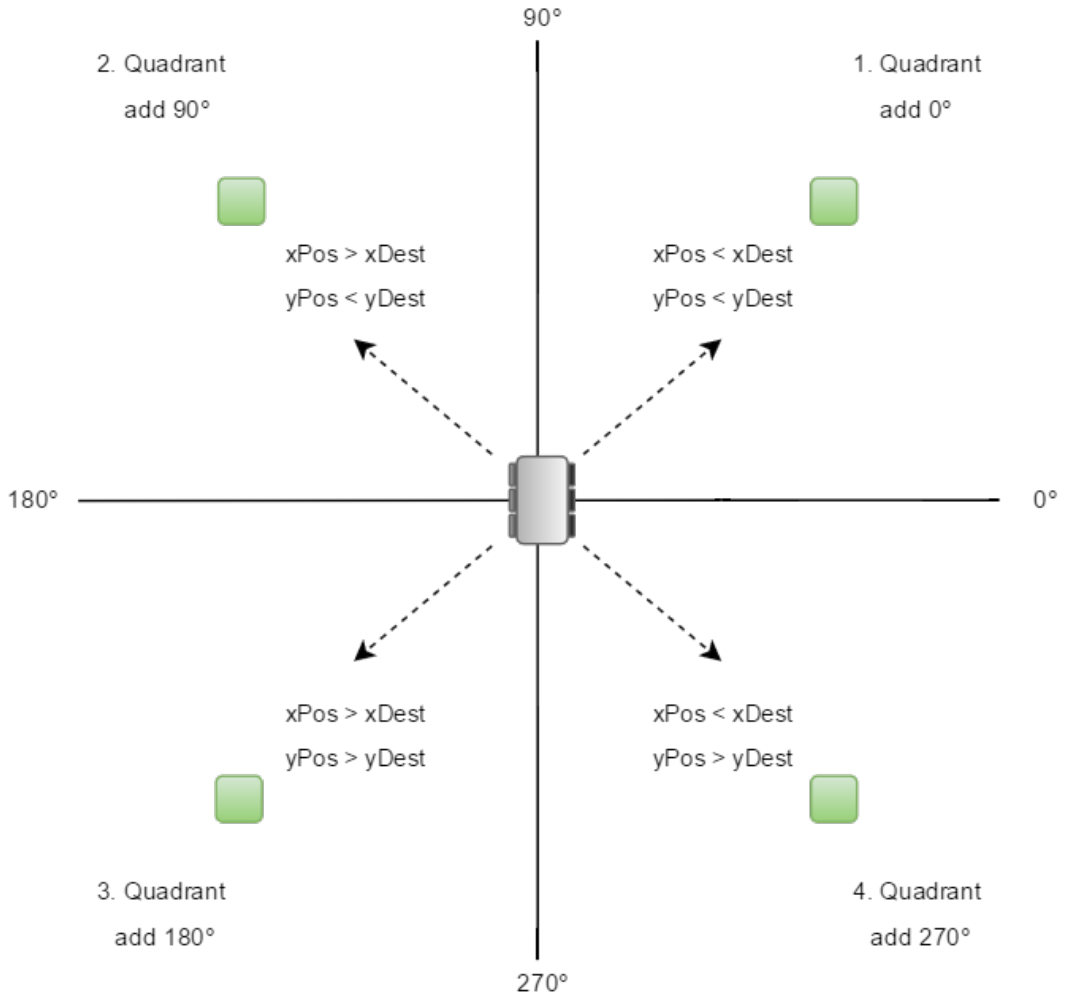

Figure 7.11: The 4 quadrants where a destination may reside, and the logic to find the optimal bearing for each quadrant

The Optimal Bearing to a destination in the first quadrant is found by adding 0° to α, in the second quadrant we add 90°, 180° in the third quadrant and 270° in the fourth. To keep the value of the optimal bearing within the 4 quadrants, the value is done modulus 360.

With the use of the Optimal Bearing, we can now find Δb the difference between the optimal bearing and the robots current bearing. This will in turn be used to calculate in which direction the robot should turn to have the shortest rotation

distance. Δb can be calculated with the following Equation:

$$\Delta b = currentBearing - optimalBearing \tag{7.8}$$

And with Δb the rotation logic in Figure 7.12 decides whether the robot is on course or not, and in which direction it should rotate to have the shortest rotation distance.

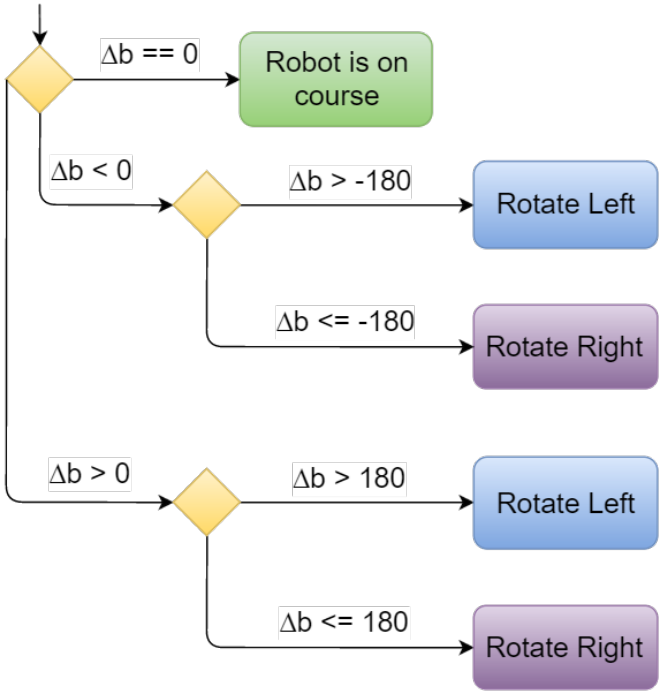

Figure 7.12: Rotation logic for the robot using Δb

7.3.5 MVP Iteration 5: Graphical Simulation

Having a graphical simulation of the robot and its destination will greatly help the development of the control algorithm and the cooperation between the modules in the system. By using a graphical simulation one can see if the robot is actually moving towards it destination, if it has reached it and if it is rotating in the correct direction without having to check log files, console output and etc.

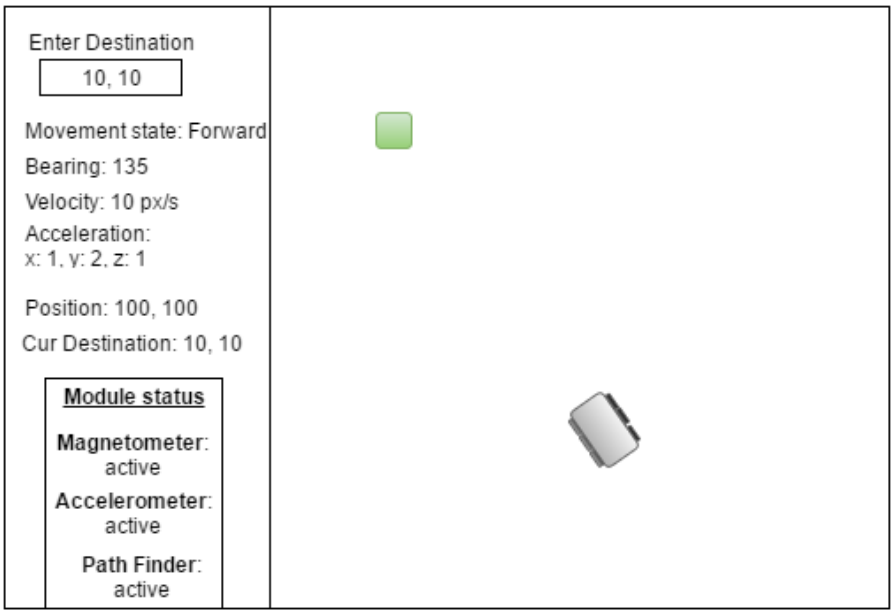

Figure 7.13: Early sketch of GUI, elements on the left showing status of the simulated robot and status of the modules in the system

Figure 7.13 contains a sketch of the graphical user interface(GUI). The goal of the GUI is to show the simulated physical properties and to draw the robot and destination on a canvas. By doing this one can check if the cooperation between the modules are working and to check if the control algorithm is working. There is also a panel showing module status, this is to show the sate of the modules in the system. OSGi modules (bundles) can have one of six different states, as shown in listing 6.2.

Part III

Results

Chapter 8

System design and implementation

This chapter describes the design process of the system, the finalized system design and the implemented prototype of a control system using the design.

When designing and developing this system, the minimum viable product technique(20) was used. The theory and research done for each MVP iteration can be found in section 7.3. The system was designed with high focus on modifiability **??**, with modules having high cohesion and low coupling. This was done to better accommodate the use of OSGi bundles in the implementation.

To describe the system, I will start with the system design as a whole and then dive into each of the top-level modules(OSGi bundles). The system was developed using the tool Reactive Blocks, see section 6.1. When creating systems using this tool, one divide the system into blocks (modules). These blocks can contain other blocks and/or java code. The result of the merging of the two technologies OSGi and Reactive Blocks is a system consisting of modules within modules, blocks within blocks.

8.1 System Design

In section 7.3 an early sketch of the system design was described and its alterations to incorporate the inter-modular communication. This sketch was used as the foundation and inspiration for the rest of the system design and implementation, and the final version of the system design can be seen in Figure 8.1

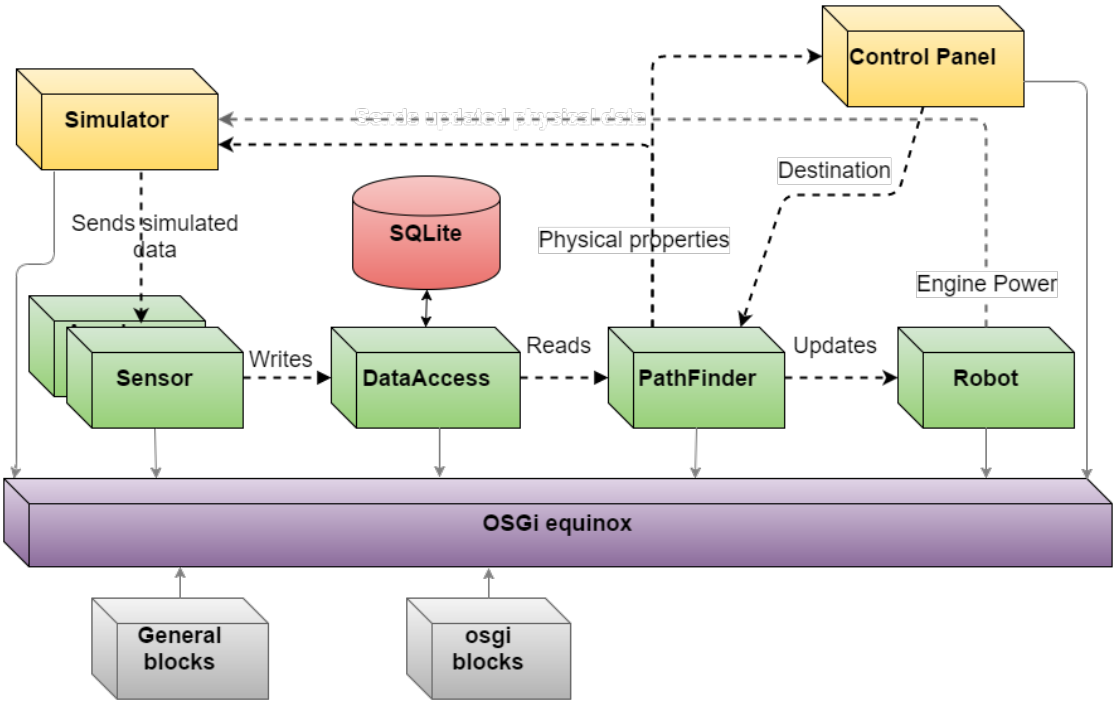

Figure 8.1: Design of the current system

In Figure 8.1 we can see several squares in different colors and one red cylinder. The squares represent independent modules registered in the OSGi framework as a part of the system, implemented as OSGi bundles. The green and yellow squares are modules with their own life-cycles and work as a part of the Simulation loop, see Figure 7.5. The gray squares are not part of the simulation loop, their job is to offer POJO classes to the other modules in the system, as well as different blocks used by the other modules. The yellow squares are squares that can be removed, if the Simulation is going to be converted to work on a real device, see Figure 8.2.

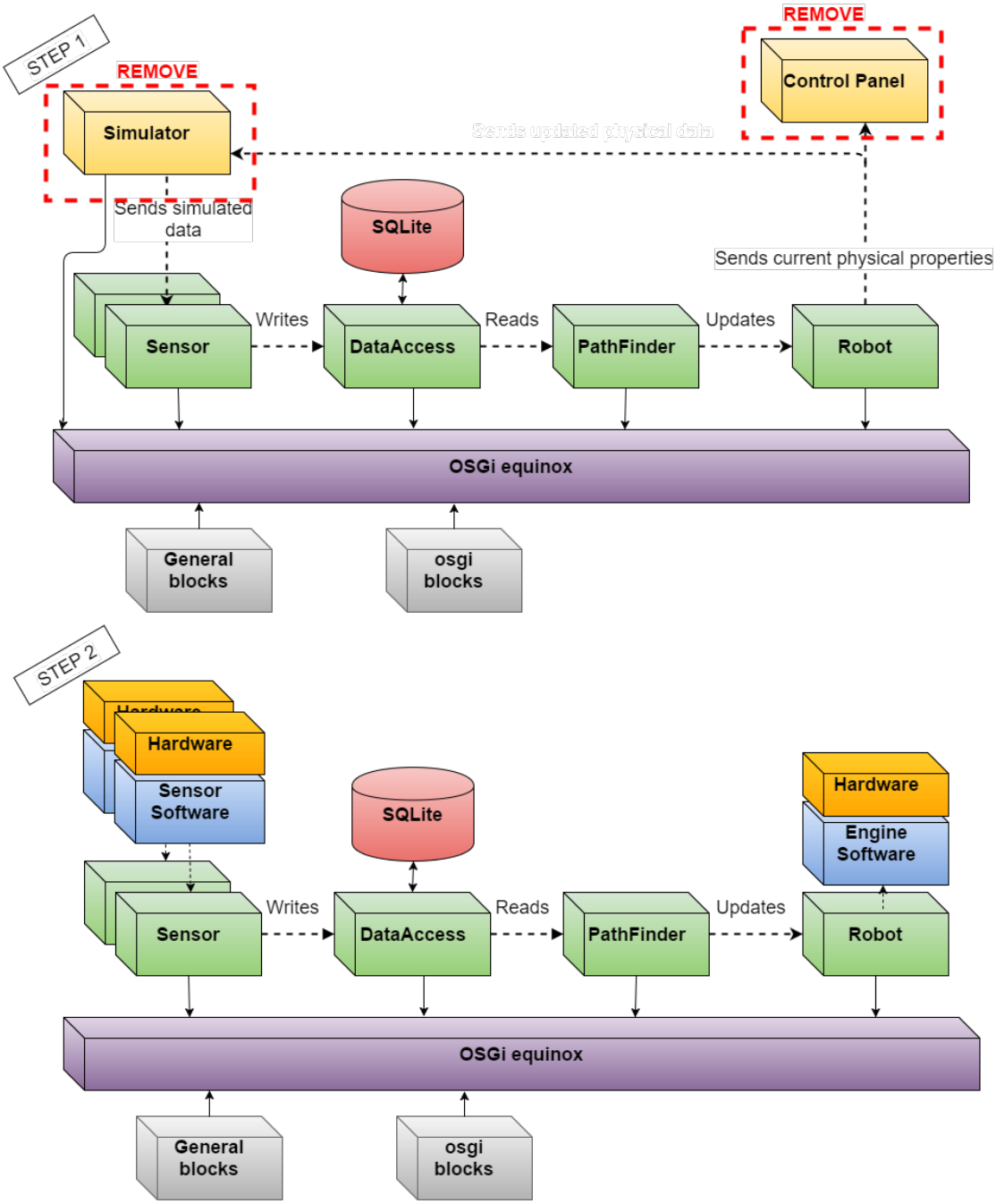

Figure 8.2: Illustration of the steps needed to convert the Simulation system to a real robot

In Figure 8.2 we can see an example of how one can convert the control software to work on a real device with 2 simple steps. All squares in the design represent

independent modules with their own life-cycles, and can be removed without any significant modification to the other modules in the system. When converting the system to a real device, the simulator module and the control panel module are not necessarily needed and can be removed as shown in *step 1.* Since the modules use a publish-subscribe pattern, see section 7.3.1, the modules are not aware of where they get their data, and which modules are subscribing to their data. Because of this we can see in step 2, there are added new modules (blue squares) containing hardware specific software. On the left we can see sensor software, specific for the sensors used in the device has been added and started to publish data on the Topic of which the sensor modules are listening. The same is done on the right side, where software specific for the engines is listening to the topic published by the robot module.
When converting the software from Simulator to a real robot, it will most likely require some modifications in the existing modules. However this work can be greatly reduced if the simulated version of the system is similar to the device it will be used on.

Figure 8.1 is a general model of the system, showing which modules exist in the current prototype of the system and which modules are required for the simulation loop to work. The next sections will be used to explain each of these top-level modules in detail, how they have been implemented using reactive blocks and the communication between them.

8.1.1 Osgi blocks Module

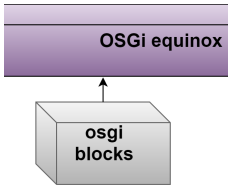

Figure 8.3: The osgiblocks module

The *osgiblocks module* is shown in Figure 8.1 as a gray block, indicating that it does not have its own life-cycle and its only purpose is to export blocks related to using the OSGi framework, to the other modules in the system. The module provides the four following blocks:

- RegisterService
- FetchService

- OsgiEventSender
- OsgiEventListener

These four blocks are responsible for the inter-modular communication related to the first minimum viable product iteration in section 7.3.1. The *OsgiEventSender* and *OsgiEventListener* handle the publish-subscribe communication described in section 7.3.1. The *RegisterService* and *FetchService* blocks are used by modules registering a service in the framework, and by modules using the service.

RegisterService block

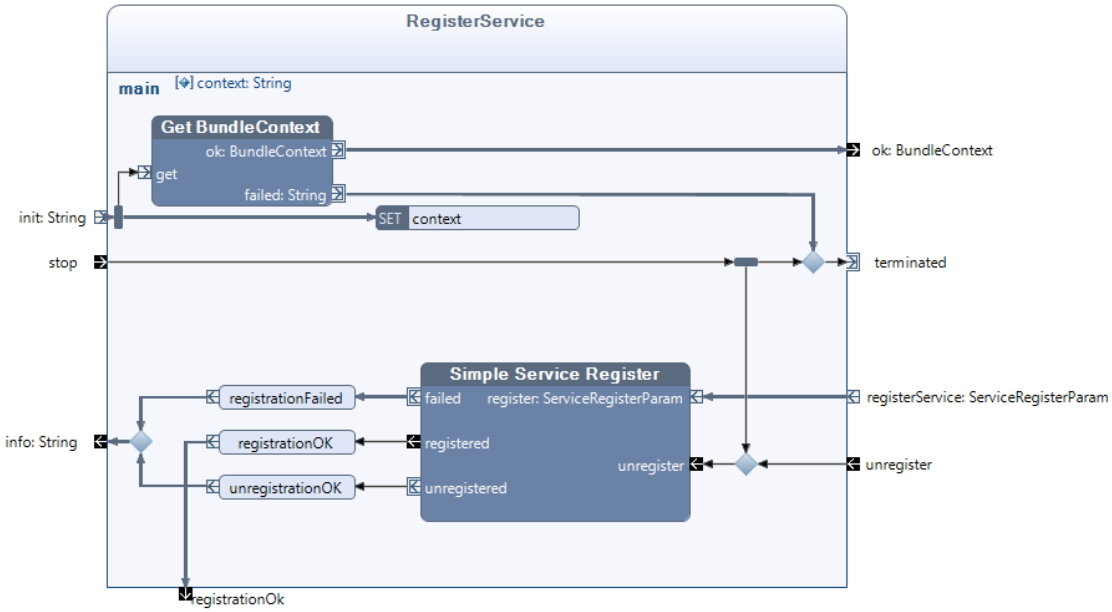

Figure 8.4: The RegisterService block

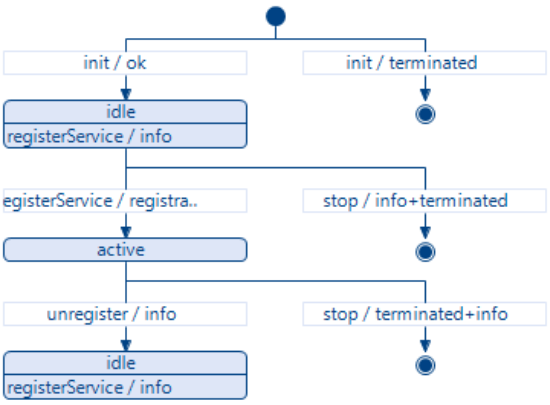

Figure 8.5: The External State Machine diagram for the block RegisterService

The RegisterService block serves as a convenient way for a top-level module (OSGi bundle) to register a service in the OSGi framework. In Figure 8.4 we can see a screen shot of the block implemented in the tool Reactive Blocks, in Figure 8.5 we can see the blocks *External State Machine diagram*. The block is started by a sending a string on the *init* input pin on the left side of the block. The string is usually the name of the context registering the service, but is of no importance to the functionality of the block. Registering the context triggers the internal *Get BundleContext* block, which retrieves the bundle context. If there is an available bundle context, the bundle context object is output on the pin *ok* and the block enters the state *idle*, see Figure 8.5. In the state *idle* the block can accept input on the *registerService* pin, this is the where the actual service registration happens. The pin requires an object of the class *ServiceRegisterParam* containing the name of the service class, the service object and optional properties. These two/three parameters are required by the OSGi framework when registering a service.

When an ServiceRegisterParam object arrives at the input pin, it is passed along to the internal block *Simple Service Register* which handles the service registration in the framework. The registration can either be successful, triggering the *registrationOk* output pin, or it can fail and trigger the info output pin with an error message posted by the *Simple Service Register* block. In Figure 8.5 we can see that an successful service registration triggers the *active* state of the block, an unsuccessful registration keeps the block in the state *idle* and outputs an error message.

FetchService block

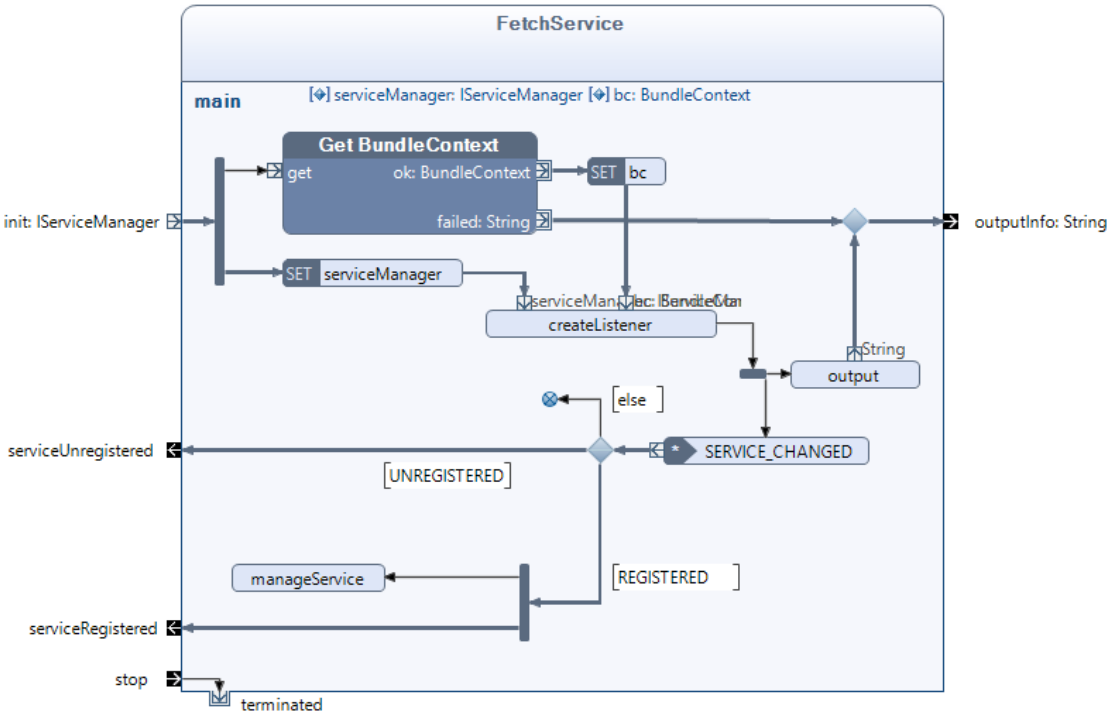

Figure 8.6: The FetchService block

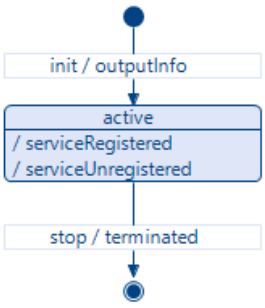

Figure 8.7: The External State Machine diagram for the block FetchService

When a module needs a registered service it can use the *FetchService* block, see Figure 8.6 for implementation in Reactive blocks and Figure 8.7 for external state

machine diagram. The *init* pin of the block requires a implementation of the interface *IServiceManager*, see code-snippet in Figure 8.8. An implementation of this interface describes what should be done with the service when the FetchService block finds it, and the name of the service which the block should look for.

```
package mstr.osgiblocks.api;

public interface IServiceManager {

    boolean manageService(Object serviceObject);

    String getServiceClassName();

}
```

Figure 8.8: Code-snippet of the interface IServiceManager

When the FetchService block receives an servicemanager object, the block retrieves the bundle context from the *Get BundleContext* block and stores the servicemanager in a local variable, see Figure 8.6. After this is done the next step is to trigger the *createListener* method, see Figure 8.9.

```
public void createListener(IServiceManager serviceManager, BundleContext bc) {
    serviceListener = new GeneralListener(serviceManager, bc, this);
}
```

Figure 8.9: The createListener method snippet

The GeneralListener class used in Figure 8.9 is an implementation of the *SerivceListener* interface provided by the OSGi specification. An implementation of this class can be registered in the bundle context as a listener, listening for a service using the service class-name. As can be seen in Figure 8.9, the constructor of the class takes three parameters; A servicemanager, bundle context(bc) and the block(this) that creates the Listener. The servicemanager is used by the listener to get the name of the service it should listen for. The bundle context is the bundle context the listener should register itself in and the block reference is used by the listener to push events to the block whenever it observes changes to the service.

In Figure 8.6 the SERVICE_CHANGED eventlistener can be seen, the eventlistener listens for events pushed by the servicelistener. The servicelistener pushes events with additional data, a string containing either registered or unregistered. If the Registered string is received, the FetchService block calls the manageService

method, as can be seen in Figure 8.6, this method calls the servicemanagers *manageService* method see Figure 8.8. See Figure 8.12 for an implementation of this functionality, what happens to the service and how it is used is up to the developer of the *ServiceManager*.

OsgiEventSender block

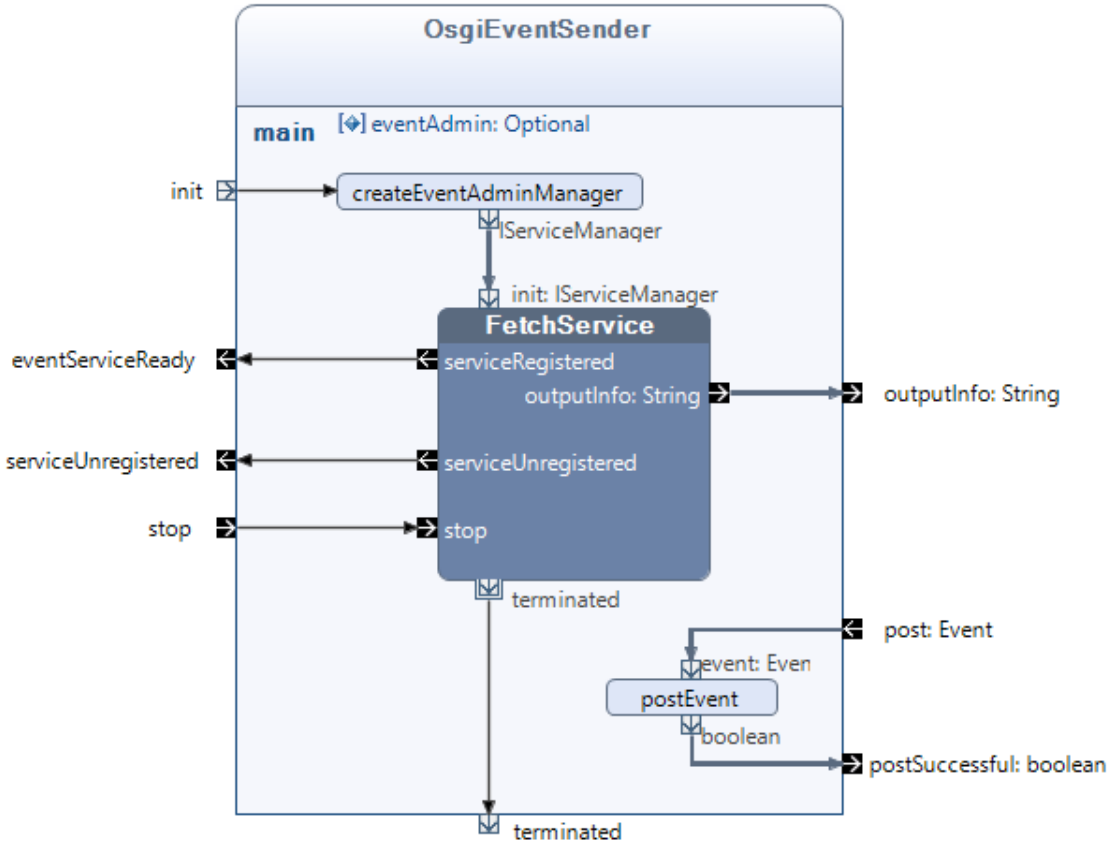

Figure 8.10: The OsgiEventSender block

The OSGiEventSender block is used by modules to send/publish events through the OSGi framework. The OsgiEventSender uses the FetchService block to fetch a service called the *Event Admin Service*. This service is shipped with the OSGi framework and is started when the OSGi framework starts. The event administrator provides a basic public-subscribe model, where each event consist of a topic and a set of properties.

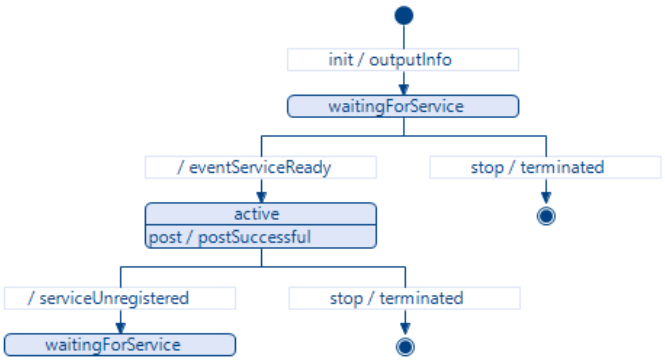

Figure 8.11: The External State Machine diagram for the block OsgiEventSender

When the input pin *init* is triggered from an external source, the first step is to run the *createEventAdminManager* method. As described under the Fetchservice section, the FetchService block requires an implementation of the IServiceManager(Figure 8.8) to be started. An implementation of this interface is instantiated in the createEventAdminManager method and passed on to the FetchService block, a code-snippet of the method *createEventAdminManager* can be seen in Figure 8.12.

```
import java.util.Optional;

public class OsgiEventSender extends Block {
    public java.util.Optional<EventAdmin> eventAdmin;

    public IServiceManager createEventAdminManager() {

        return new IServiceManager() {

            @Override
            public boolean manageService(Object serviceObject) {
                if (serviceObject instanceof  EventAdmin) {
                    logger.debug("FOUND EVENT ADMIN SERVICE");
                    eventAdmin = Optional.of((EventAdmin)serviceObject);
                    return true;
                }
                logger.debug("COULD NOT FIND EVENT ADMIN SERVICE");
                return false;
            }

            @Override
            public String getServiceClassName() {
                return EventAdmin.class.getName();
            }

        };
    }
```

Figure 8.12: Code-snippet from the OsgiEventSender block, showing the createAdminEventManager method

The OSGiEventSender block enters the *waitingforservice* state after it is initiated and stays here until the FetchService block finds the event admin service. When

this happens the block triggers the eventServiceReady pin, and the block enters state *active*. In the active state the block can now accept input on the *post pin*, which calls the postEvent method. This method uses the event admin service to post an event on the OSGi framework. The OSGiEventSender will stay in the *active* state, accepting events, until it is stopped through the stop input-pin or the FetchService notifies the block that the service has become unavailable. If this happens the block will go back to the *waitingforservice* state. The events posted by the event admin service is a data and value pair wrapped in the *Event* class provided by the OSGi specification (23).

OsgiEventListener block

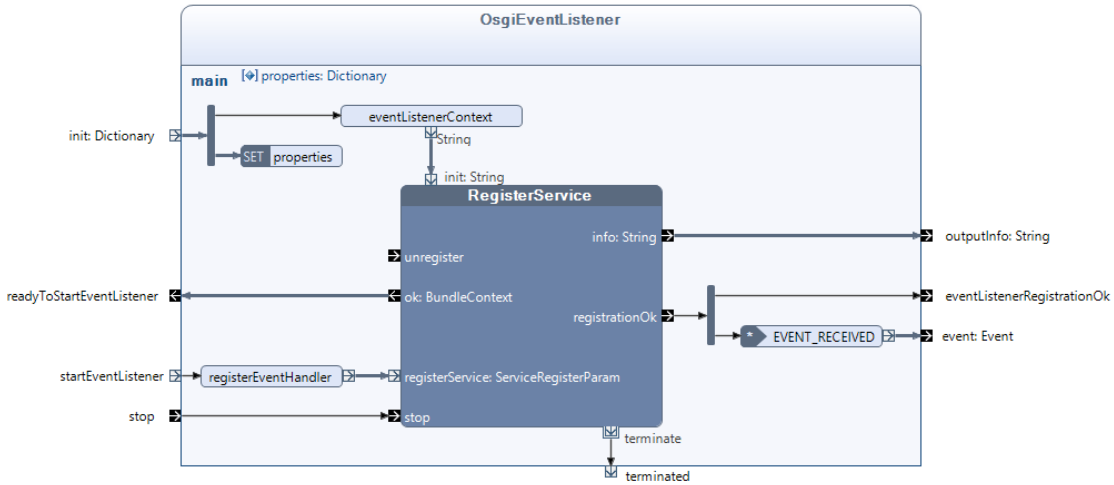

Figure 8.13: The OsgiEventListener block

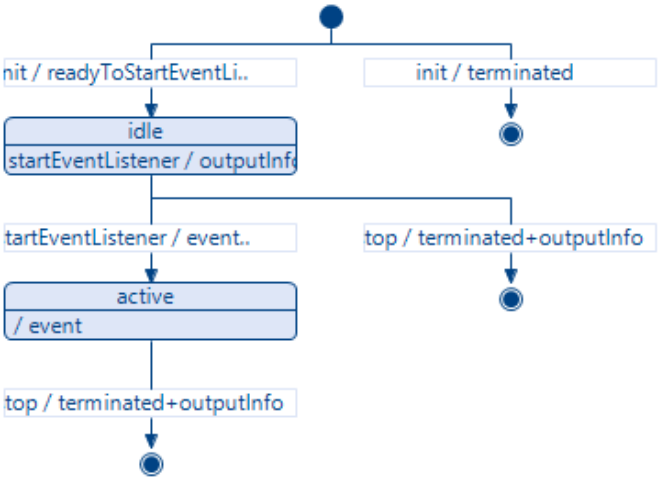

Figure 8.14: The External State Machine diagram for the OsgiEventListener block

The OSGiEventListener block is used by modules in the system to listen for events published under a topic. Listening to events in the OSGi framework requires the use of a sub-class of the abstract class *EventHandler* (24). This class is shipped with the OSGi framework and contains 1 abstract method, the handleEvent method. Inside this method developers must specify what the eventhandler should do in-case it receives an event on the specified topic, see Figure 8.15 for code-snippet. The eventhandler is then registered as a service in the OSGi framework.

```
public ServiceRegisterParam registerEventHandler() {
    EventHandler eventHandler = new EventHandler() {
        @Override
        public void handleEvent(Event event) {
            sendToBlock("EVENT_RECEIVED", event);
        }
    };
    return new ServiceRegisterParam(EventHandler.class.getName(), eventHandler, properties);
}
```

Figure 8.15: Code-snippet from the OsgiEventSender block, showing the registerEventHandler method

In Figure 8.13 we can see the implementation of the block in Reactive Blocks. The block is initiated by the *init* pin, which requires a Dictionary instance. The dictionary should contain the topics the event handler should listen for. After receiving a dictionary, the block starts the inner-block RegisterService and enter the state *idle*. The block stays in this state until it is triggered by the *startEventListener* input-pin. This pin triggers the registerEventHandler method and sends

the created event handler to the RegisterService block. When the event handler has been registered in the framework the RegisterService block pushes a token onto the *eventListenerRegistrationOk* pin, see Figure 8.13. When the eventlistener is registered, it can start pushing events it find in the framework out to the OsgiEventListener block. As can be seen in Figure 8.15 the event listener sends events it finds in addition to the keyword EVENT_RECEIVED to the block. This gets picked up by the event listener EVENT_RECEIVED shown in Figure 8.13 and pushed out on the output-pin *event*.

Osgi blocks Module Summarized

Throughout this section the four OSGi specific blocks exported by the Osgiblocks module have been explained in detail. The RegisterService and FetchService blocks can be used to register and fetch services from the OSGi framework as a way for modules in the system to share their functionality. The OsgiEventSender and OsgiEventListener blocks were described as blocks with the possibility to publish and subscribe to events using the OSGi framework. These blocks provide a convenient way to achieve inter-modular communication with a small amount of binding between the modules.

The next section will describe the other "gray" module of the system, the *General Blocks* module.

8.1.2 General blocks Module

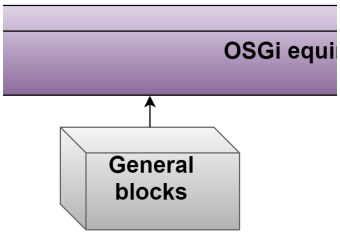

Figure 8.16: The General blocks module

The *General blocks* module is shown in Figure 8.1 as a gray block, the same as the Osgi blocks described in the last section. It is gray because it does not have a lifecycle, only exporting blocks, classes and system wide constants to other modules in the system. The module export one block (sub-module), the SensorBlock which

will be explained in detail later in this chapter. It also exports a package with model-classes used by the other modules in the system. The package contains the classes shown in Figure 8.17

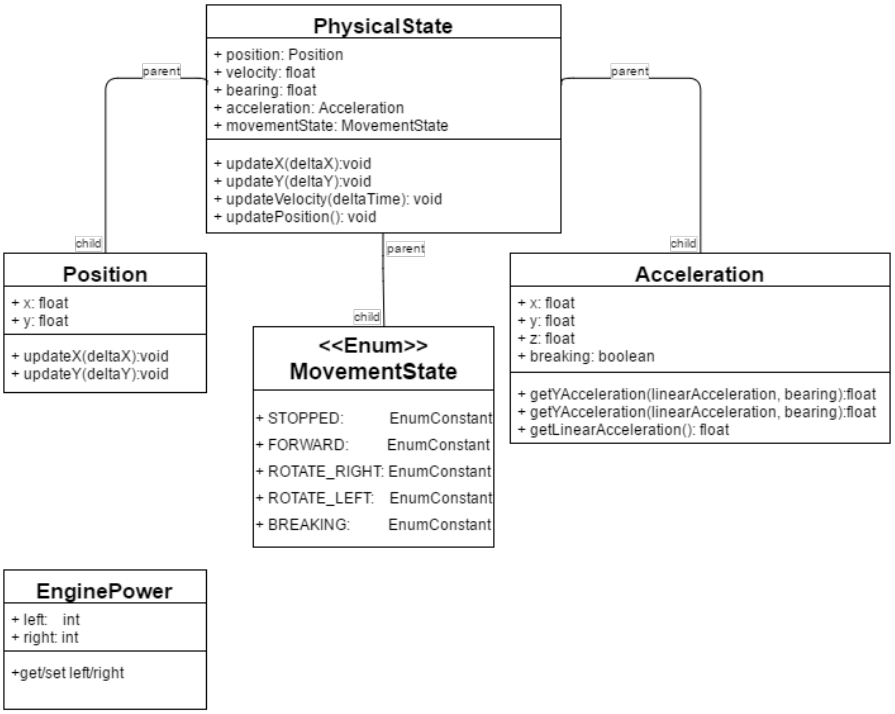

Figure 8.17: Class diagram of the the *General Blocks* modules exported classes

The classes exported by the General Blocks module contain 5 classes. The *PhysicalState* class, which contains all the physical properties of the robot. The *Position* and *Acceleration* classes are used to store information about the position and acceleration of the robot, the *MovementState* enum is used to describe which movement state the robot is in and The last class *EnginePower* is used to store the current engine power of the robots engines.

```
public class RobotConstants {
    //Constants for magnetometer simulation
    public static final String MAGNETOMETER_DATA_TOPIC = "mag_simulation";
    public static final String MAGNETOMETER_EVENT_NAME = "bearing";

    //Constants for acceleration simulation
    public static final String ACCELEROMETER_DATA_TOPIC = "acc_simulation";
    public static final String ACCELEROMETER_EVENT_NAME = "bearing";

    //Robot constants
    public static final float MAX_DEGREES_PR_SECOND = 23f; // degrees/s
    public static final float MAX_ACCELERATION_PR_SECOND = 1f; // px/second
    public static final float MAX_DECELERATION_PR_SECOND = -4f; // px/second
    public static final int MAX_SPEED = 8;// px/second
```

Figure 8.18: Code-snippet from the RobotConstants class, contained in the General Blocks module.

In addition to export classes, the module contains the Systems various constant variables. In Figure 8.18 a code snippet from this constant class can be seen. The constants seen in the code-snippet are the constants used as topic for the communication between the simulator and the two implemented sensors, as well as constants limiting the speed, rotation speed, acceleration and deceleration of the simulated robot.

SensorBlock

The only block exported by the General Blocks module is the SensorBlock. When creating a sensor module, there were some operations and blocks that always had to be set up in the same manner, the work was time-consuming and complex. To battle this work and to stay true to the DRY(Don't Repeat Yourself) principle, the SensorBlock was created. The block provide the following functionalities to a sensor module:

- Registering the sensor in the database
- Registering measurements in the database
- Selecting data from the database

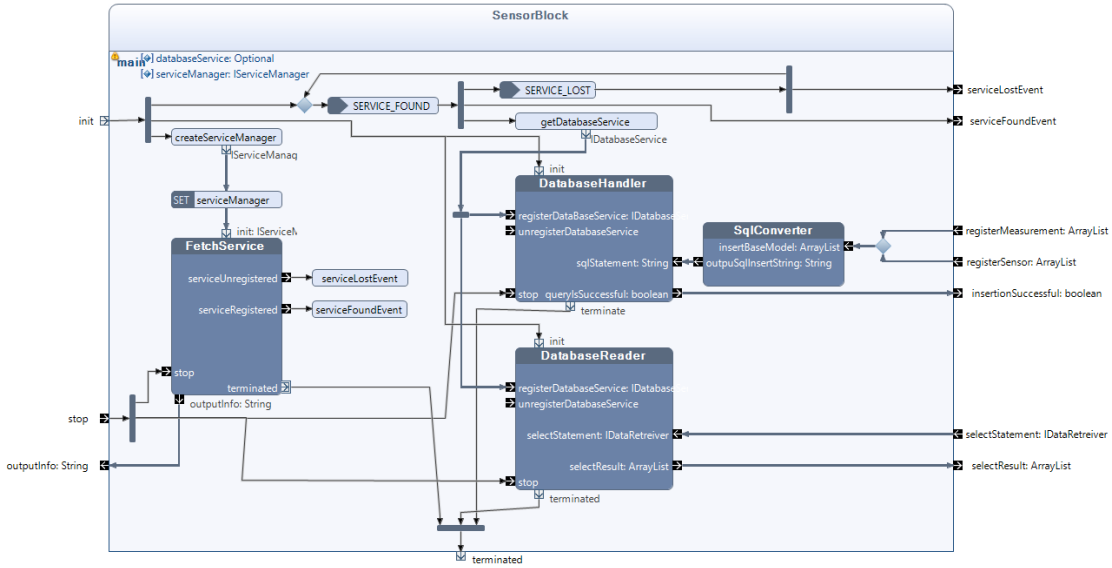

Figure 8.19: The SensorBlock

When the SensorBlock is initiated, it triggers four operations in parallel. The first is to create an serviceManager object, used by the FetchService to find a databaseService, see the *createServiceManager* method in Figure 8.19. The databaseService is created and published by the DataAccess module to access the database. The serviceManager is sent to the FetchService module and starts listening for a databaseService in the OSGi framework. In addition to this the block *DatabaseHandler* and *DatabaseReader* is initiated and the event listener SERVICE_FOUND is started. When all these steps have been concluded, the SensorBlock enters the state *waitingForService*, see Figure 8.20

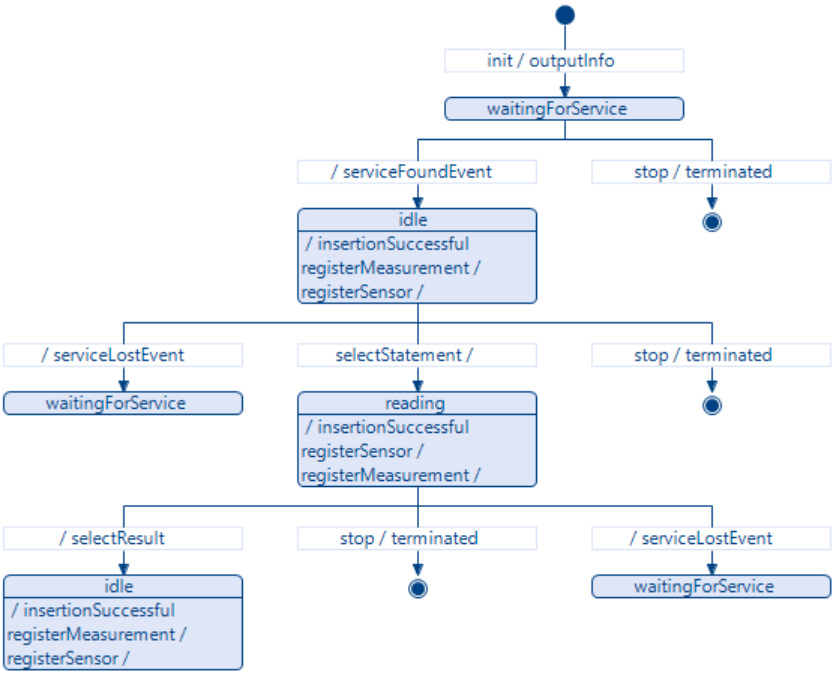

Figure 8.20: The SensorBlock External State Machine diagram

When the FetchService block finds the databaseService it pushes a notification on its serviceRegistered output-pin and triggers the operation *serviceFoundEvent*. The serviceFoundEvent publishes an event to the block with the keyword SERVICE_FOUND, and triggers the operation *getDatabaseService* which gets the databaseService found by the FetchService block and sends it to the DatbaseHandler and the DatabaseReader blocks. When this is done the SensorBlock enters the *idle* state, the block accepts input on the *registermeasurement*, *registerSensor* and *selectStatement* pins. Input on the *selectStatement* pin will trigger the state reading, in which the block will wait for data read from the database. When the databaseService enters the reading step, see step 1 in Figure 7.6, the output-pin *selectResult* is triggered with the results from the select statement and the block re-enters the *idle* state. The functioanlities of the three database-specific blocks used by the SensorBlock in Figure 8.19 will be explained in detail later in this chapter.

General Blocks module Summarized

The general blocks export a package of classes used by the system to store the robots physical properties about the , Figure 8.17. It provides the static class *RobotConstants*, containing constants used by the other modules in the system, Figure 8.18. The module also export one block, the SensorBlock, Figure 8.19. This block provides a convenient and reusable way for sensor modules to get access to the database.

In the next section the DataAccess module will be described, a module with its own life-cycle the first module that is a part of the *simulation Loop*.

8.1.3 DataAccess module

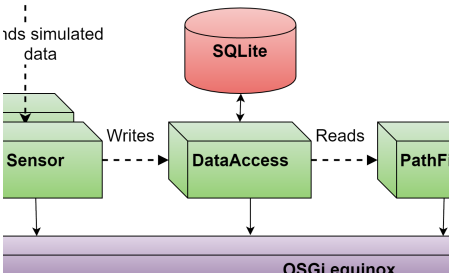

Figure 8.21: The DataAccess module

The *DataAccess* module was created to place the responsibility of the database communication in its own module. The module has its own life-cycle in the OSGi-framework, this means that it is a running process with the ability to exist in one of the life-cycle states listed in listing 6.2. The module has the responsibility of creating the SQLite database, providing a service for the other modules in the system to communicate with the database and exporting blocks that can be used to "wrap" the database service easing the use of the service itself.

In addition to the databaseService and the database specific blocks, the module exports a package containing classes specifically designed to fit the data-model used by the database. In Figure 8.22 a class diagram of these classes can be seen.

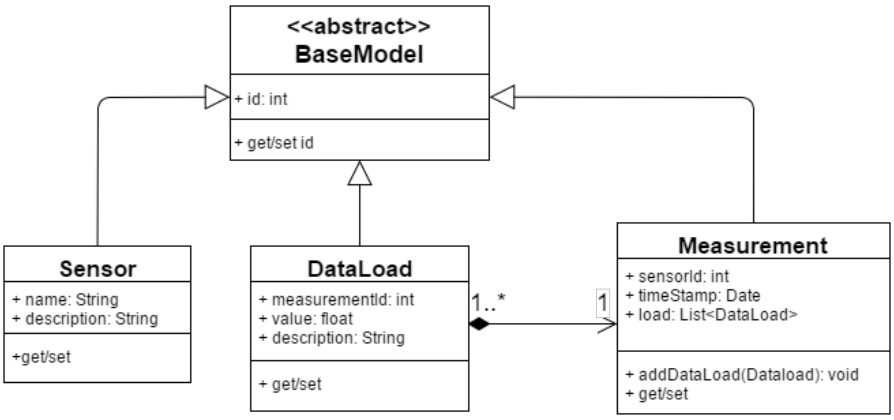

Figure 8.22: Class diagram showing classes exported by the DataAccess module.

The classes shown in Figure 8.22 are used by the modules to store data related to sensor measurements. The classes were created to correspond with the tables in the database ER-diagram, see Figure 8.23

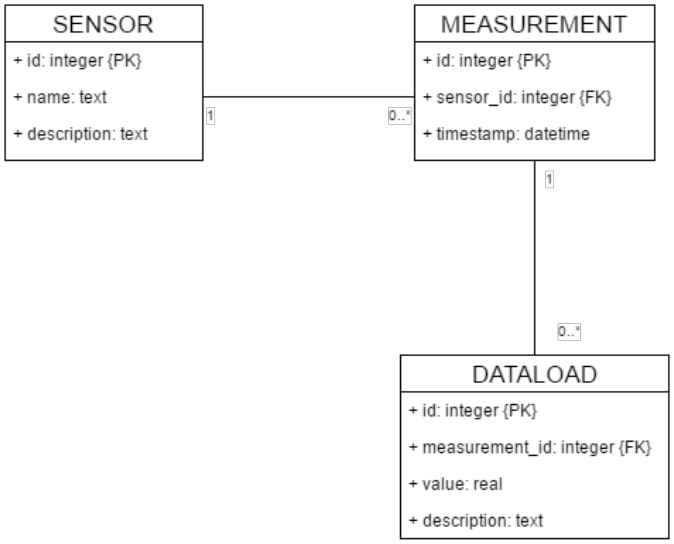

Figure 8.23: ER-diagram of the SQlite database created by the DataAccess module

The database was made as simple as possible, but yet flexible enough to accept most types of floating value sensor measurements. An measurement can consist of several dataloads and each of these dataloads can contain a value and a description.

DataAccess block

The DataAccess *application block* is the block containing the life-cycle of the DataAccess module, and is the part of the module that is *Run* by the OSGi framework. The Reactive Block implementation can be seen in Figure 8.24

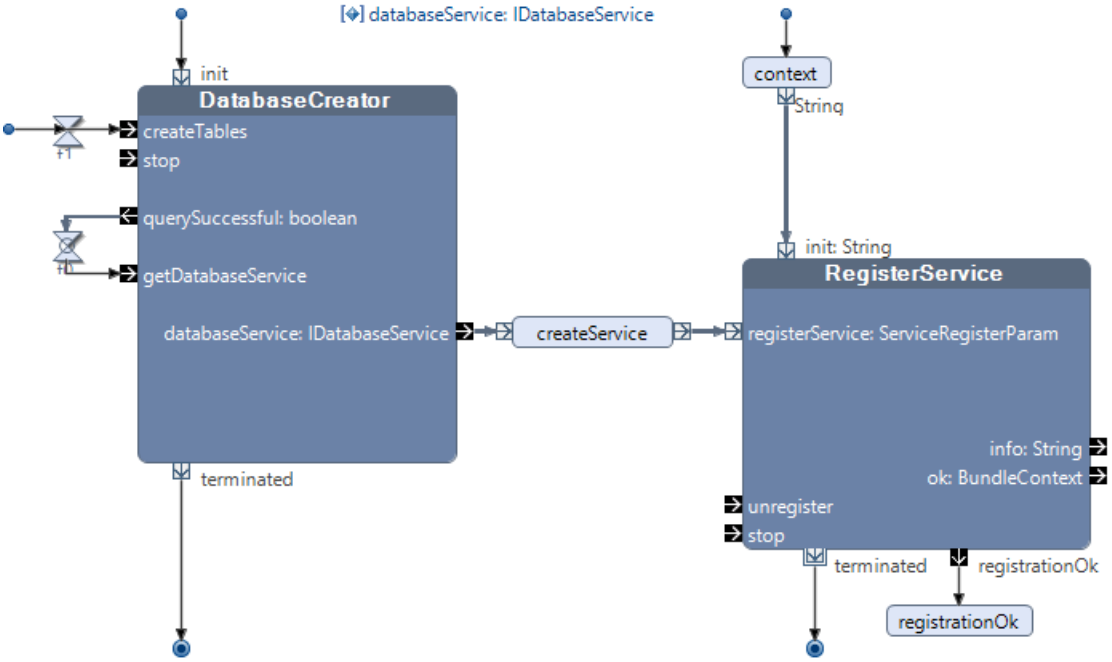

Figure 8.24: The DataAccess block containing the life-cycle components of the DataAccess module, implemented in Reactive Blocks

As can be seen from Figure 8.24, the DataAccess block itself is not advanced. The block has three initiation pins (small blue circles) which starts in paralell when the block is initiated. The block is started with initiating the *DatabaseCreator* and the *RegisterService* block. The last initiation pin starts a timer (100ms) to let the DatabaseCreator get fully started, before it triggers the input-pin *createTables*. This pin triggers the creation of the database's tables, and after it is done it notifies the *querySuccessful* pin. This in turn triggers the *getDatabaseService* pin which results in output on the outgoing *databaseService* pin. The output from this pin is the newly created DatabaseService, explained in detail in the next paragraph under *The DatabaseService class*. This newly created DatabaseService object is then wrapped in a *ServiceRegisterParam* object by the *createService* method and sent to the RegisterService block for registration in the OSGi framework.

The DatabaseService class

The DatabaseService class is an important part of the system, and is created in the DatabaseCreator block. The service is then registered in the OSGi framework and used by all Simulator modules and the PathFinder module to communicate with the database. The DatabaseService class contains the core purpose of the DataAccess module and can in some way be viewed as the *Pulse* of the system, this because it is this service which controls the 2 step read/write functionality described in Figure 7.6. The DatabaseService implements the interface *Runnable* and runs on its own thread. The thread runs the two step read/write pulse, from now on called the *simulation Pulse*. In Figure 8.25 a classdiagram of the DatabaseService can be seen, with all its fields and methods.

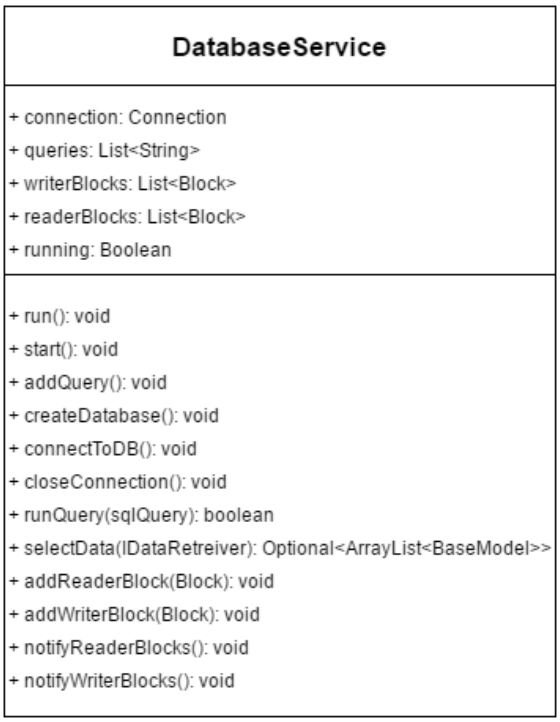

Figure 8.25: Classdiagram of the DatabaseService class, showing its fields and methods.

When a module/block wants to use the databaseservice, it has to do the following operations; If the module is using the to write data, it registers its query in the *queries* field(this field works as the buffer described in Figure 7.6), and registers itself in the *writerBlocks* field. If a module is using the service to read data, it registers itself in the *readerBlocks* field. The *simulation Pulse* can be seen in the

code-snippet in Figure 8.26

```
@Override
public void run() {
    while(running) {

        String query = createQueryString();
        queries = new ArrayList<String>();
        boolean succ = false;

        if(!query.isEmpty()){
            succ = runQuery(query);
        }

        notifyWriterBlocks(succ);
        notifyReaderBlocks();

        try {
            Thread.sleep(PULSE_TIMEOUT);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

private synchronized void notifyWriterBlocks(boolean success) {
    writerBlocks.forEach(block -> block.sendToBlock("QUERY_FINISHED"));
    writerBlocks = new ArrayList<Block>();
}

private synchronized void notifyReaderBlocks() {
    readerBlocks.forEach(block -> block.sendToBlock("READ_READY"));
    readerBlocks = new ArrayList<Block>();
}
```

Figure 8.26: Code snippet from the DatabaseService runnable loop (*Simulator Pulse*)

The first step of the *simulation pulse* is to create an query string, this is done by combining all the insert statements registered in the buffer (queries list). The pulse then checks if there actually were any queries, and if there were, stores them in the database. The success of the query, either true or false, is stored in the *succ* variable and sent to all the writer blocks by using the *notifyWriterBlocks* method. The next step is to notify all the reader blocks with the method *notifyReaderBlocks*, that the database connection now can be used to read data. This 2 step pulse is made this way because the SQlite database only accepts concurrent reading, not concurrent writing. When writing data to the database, the database is locked. The time between each write step is decided by the amount of milliseconds registered in the variable PULSE_TIMEOUT, the thread is put to sleep to allow the reading blocks some time to read and the writer blocks some time to register queries in the buffer.

With basic understanding of how the DatabaseService class works, we can now move on to the blocks that implement its functionality. There are two blocks doing this, the *DatabaseHandler* for writing data, and the *DatabaseReader* for reading data.

DatabaseHandler block

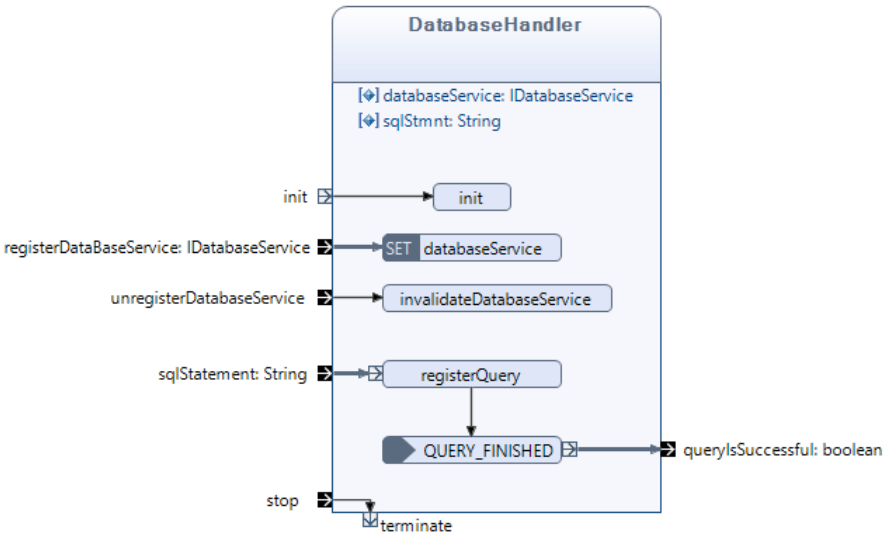

Figure 8.27: The DatabaseHandler block

The *DatabaseHandler* block provides an interface on-top of the databaseService for modules writing data to the database. The block is initiated by triggering the *init* pin, and sets the block in the *waitingForDatabaseService* state. The block exist in this state until it gets an instance of the DatabaseService class on the *registerDatabaseService* pin, this triggers the transition into the *active* state, see Figure 8.28 for external state machine diagram.

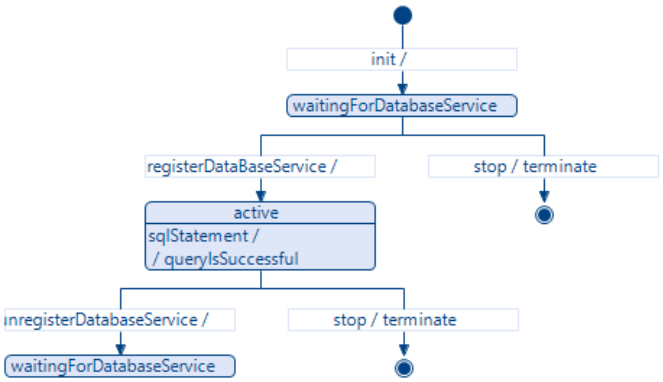

Figure 8.28: The External State Machine diagram for the DatabaseHandler block

It is in the active state that the block can be used to register data in the query buffer, see Figure 7.6. The *registerQuery* method stores the query in the buffer, and registers the block in the DatbaseService writerBlocks list. The token then rests in the QUERY_FINISHED event listener, waiting for the DatabaseService to notify the block that the query is finished, using the method *notifyWriterBlocks* shown in Figure 8.26

DatabaseReader block

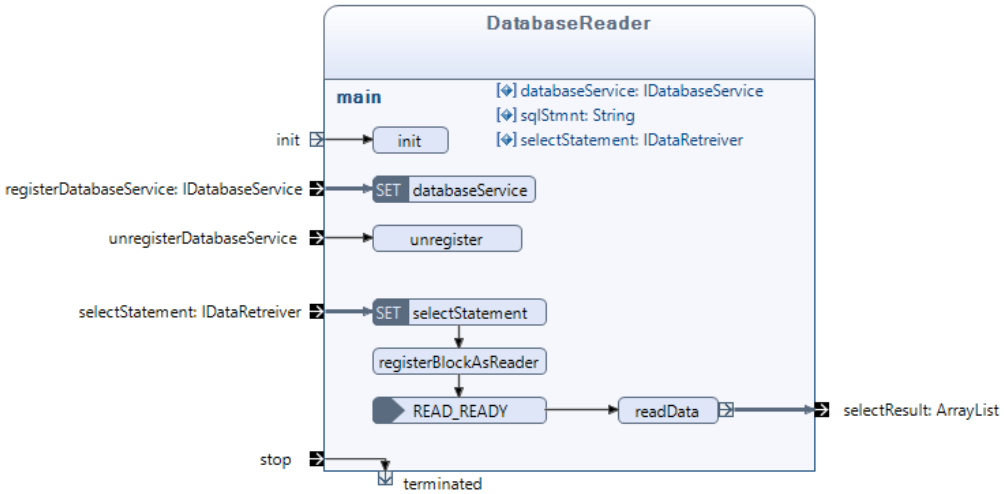

Figure 8.29: The DatabaseReader block

The *DatabaseReader* block provides an interface on-top of the DatbaseService class for reading data from the database. It is initiated by triggering the *init* pin, which moves the block to the *waitingforDatabaseservice* state. In the same way as the *DatabaseHandler*, the *DatabaseReader* block stays in the this state until it receives an DatabaseService object on the *registerDatabaseService* input-pin.

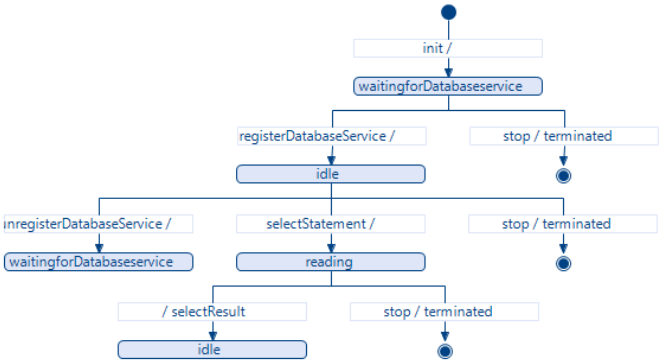

Figure 8.30: The External State Machine diagram for the DatabaseReader block

Receiving a DatabaseService object triggers the transition to the *idle* state. The block will stay in this state until the selectStatement input-pin is triggered, containing an implementation of the *IDataRetriever* interface. This action will change the blocks state to *reading*, call the method *registerBlockAsReader* and start the READ_READY event listener. The block will stay in the *reading* state until it receives an READ_READY event from the DatbaseService. Upon receiving this event the block calls the *readData* method, outputs the data it receives from the database and changes state back to *idle*.

To keep the DatabaseReader general the *IDataRetriever* interface was created. The interface decouples the select statement from the DatabaseReader and moves the responsibility to the block using the DatabaseReader.

```
public interface IDataRetriever {
    ArrayList<BaseModel> createDataSetFrom(ResultSet resultSet);
    String getSelectStatement();
}
```

Figure 8.31: Code-snippet of the IDataRetriever interface

In Figure 8.31 the IDataRetriever interface can be seen, and Figure 8.32 shows how the DatabaseService use the IDataRetriever when running an select statement.

```java
@Override
public Optional<ArrayList<BaseModel>> selectData(IDataRetreiver dataRetreiver) {
    Statement stmt = null;
    connectToDB();
    Optional<ArrayList<BaseModel>> result;
    try {
        stmt = c.createStatement();
        ResultSet rs = stmt.executeQuery(dataRetreiver.getSelectStatement());
        result = Optional.of(dataRetreiver.createDataSetFrom(rs));
        stmt.close();
        messages.add("SuccessFully ran Query: \n" + dataRetreiver.getSelectStatement());
    } catch ( Exception e ) {
        closeConnection();
        result = Optional.empty();
        messages.add(e.getClass().getName() + ": " + e.getMessage());
    }
    closeConnection();
    return result;
}
```

Figure 8.32: Code-snippet from the DatabaseService *selectData* method, outlining how the service uses the IDataRetriever to get the select statement and translating the resultSet in accordance to the IDataRetriever

DatabaseHandler and DatabaseReader summarized

In the two previous sections we have seen the details of how the DatabaseHandler block can be used to write data , and how the DatabaseReader is used to read data from the database. These blocks are exported by the Dataaccess Module and can be used by other modules in the system. The DatabaseCreator block however is only used by the DataAccess module to create the database and the DatabaseService.

DatabaseCreator block

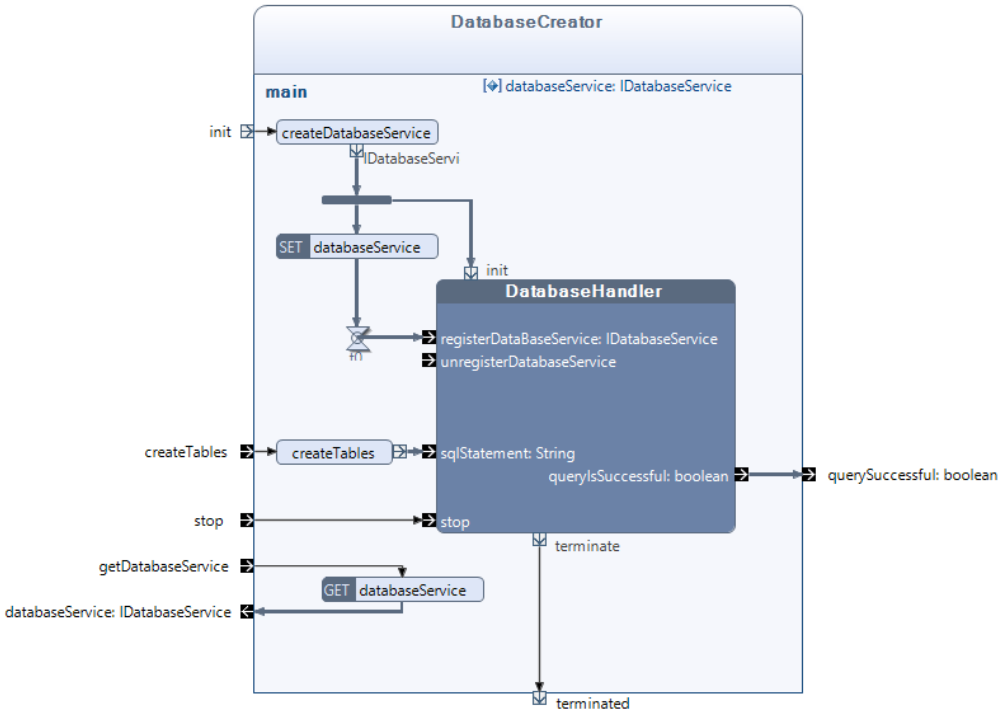

Figure 8.33: The DatabaseCreator block

The DatabaseCreator block was created to gather all the parts concerned with creating the SQLite database and the DatabaseService in one place. The block is initiated with triggering the *init* pin, which in turn calls the method *createDatabaseService*. The method creates an instance of the DatabaseService class and serves it to the DatabaseHandler. When all this is done the DatabaseHandler exists in the *idle* state awaiting input on the *createTables* pin.

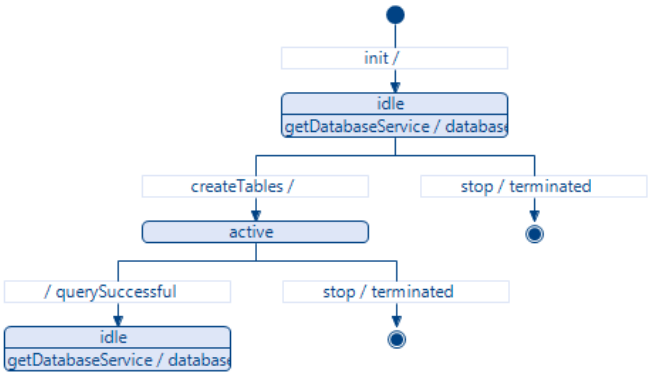

Figure 8.34: The External State Machine diagram for the DatabaseCreator block

Input on the *createTables* pin calls the *createTables* method, which generates the SQL query needed to create all the tables and their relations in the database. During the insertion of the SQL query, the block will exist in the *active* state, and stay here until the DatabaseHandler is finished running the query. When it is finished the outgoing *querySuccessful* pin is triggered and the block switches back to the *idle* state. It is now possible for external triggers on the *getDatabaseService* pin, to get the DatabaseService. This is done in the DataAccess block shown in Figure 8.24.

SqlConverter block

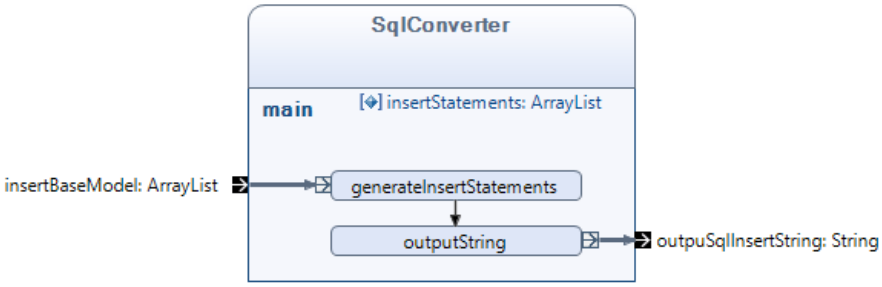

Figure 8.35: The SqlConverter block

The SqlConverter block is a convenience block created to stay true to the DRY principle. The block accepts all objects created from sub-classes of the abstract class Basemodel, shown in the class diagram in Figure 8.22. It accepts individual objects, hierarchical object structures and lists of objects and converts these into

SQL insert statements, which are output as one String on the *outputSqlInsertString* pin.

DataAccess module summarized

The Dataaccess module creates the SQLite database and the DatabaseService used by the other modules in the system. It registers the databaseService in the OSGi framework and it exports blocks such as the *DatabaseReader* and *DatabaseHandler* which can be used to access the functionality of the service. The module also export a package containing *Basemodel* classes which are used to store sensor information and sensor measurements. These classes directly reflect the table structure of the database, and can be used to contain data to and from the database. Objects of these classes can be translated to insert statements using the convenience block *SqlConverter* which translates POJO's(Plain Old Java Objects) to SQL insert statements.

In the next section the two implemented Sensor modules will be discussed. The two implemented sensors are, as mentioned earlier, the *Magnetometer* and the *Accelerometer*.

8.1.4 Magnetometer and Accelerometer modules

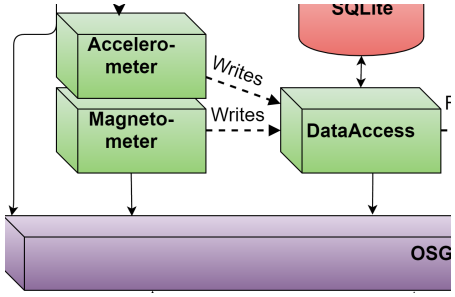

Figure 8.36: The Sensor modules

The implemented sensor modules in the system are not as complex as some of the other modules. Their job is to receive data from the Simulator module and store this in the database using the databaseService provided by the DataAccess module. The modules have their own life-cycles, and as such are implemented as application blocks in the tool Reactive Blocks. They do not export any blocks or packages and does not register any services in the OSGi framework.

The next section will explain how the modules are implemented in Reactive Blocks, and the differences between the two sensor blocks.

Magnetometer and Accelerometer blocks

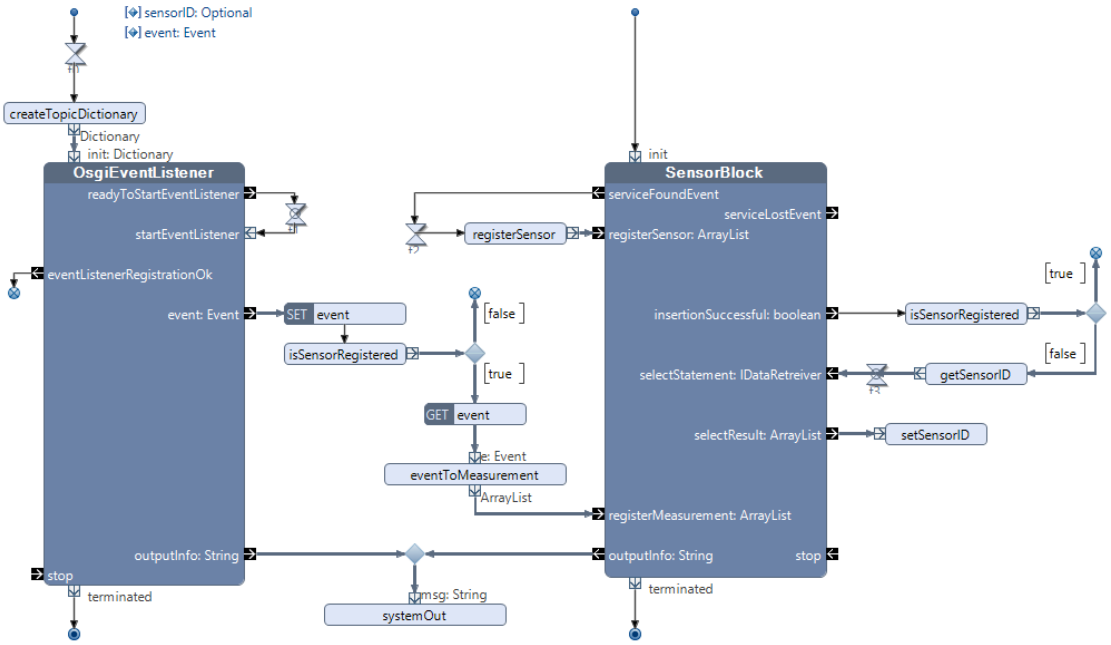

Figure 8.37: The Accelerometer and Magnetometer blocks

The sensor blocks are just a combination of the *OsgiEventListener* block from the OsgiBlocks module, see section 8.1.1 and the *SensorBlock* from the GeneralBlocks module, see section 8.1.2. The application block for both modules can be seen in Figure 8.37.

The module is started by initiation of the OSGiEventListener and the SensorBlock. The OSGiEventListener block requires a TopicDictionary to be started, which it gets from the *createTopicDictionary* method.

```
public Dictionary<String, String> createTopicDictionary() {
    Dictionary<String, String> topics = new Hashtable<String, String>();
    topics.put(EventConstants.EVENT_TOPIC, RobotConstants.MAGNETOMETER_DATA_TOPIC);
    return topics;
}
```

Figure 8.38: The *createTopicDictionary* method used when initiating the OSGiEventlistener in the Magnetometer module

In the code-snippet we can see the value used as topic is stored in the static class *RobotConstants*, exported by the *General Blocks* module, see Figure 8.18. After receiving the topic-dictionary, the OSGiEventlistener starts listening for events published under the topic, and publishes these events on the *event* output pin. When an event is found, it is stored in the global variable *event* before the *isSensorRegistered* method is run. This method checks if the sensor has been registered in the database. If it has not, the event is deleted. If the sensor has been registered the data found in the event object is translated to a measurement object and then sent to the *SensorBlock* on its *registerMeasurement* input pin.

In parallel to starting the OSGiEventListener, the SensorBlock is also started. This block requires no initial input, but after it has gotten access to the DatabaseService it triggers the *serviceFoundEvent* output pin, which in turn triggers the registerSensor method. This method creates an object of the Sensor class, see class diagram in Figure 8.17. The Sensor object created in the registerSensor method is then sent to the SensorBlock and stored in the database. When the sensor has been stored it triggers the *insertionSuccessful* output pin which in turn calls the isSensorRegistered method.

This method is created to solve two problems:

- When registering a sensor for the first time, the sensorID in the database is unknown. This because the ID field in the Sensor Table is auto incremented. This means that when the sensor is registered the first time, we have to query the database to get the sensorID value. This is needed to be able to store measurements in the database
- When registering measurements in the database, after we have registered the sensor we no longer need to get the sensorID from the database. We only need to query for the ID one time.

The problem arises because both the *registerSensor* and the *registerMeasurements* work the same way internally in the SensorBlock and both trigger the *insertionSuccessful* output pin. To battle this issue, the *isSensorRegistered* method will only return false when the Sensor has been registered in the database for the very first time. When returning false, the *getSensorID* method is called, which asks the database for the last registered sensor id. This results in the *selectResult* output pin being triggered containing the ID of the newly registered sensor, and by using the *setSensorID* storing the ID in the global field sensorID.

Difference between the Sensor modules

The only difference between the Magnetometer and Accelerometer modules are in the topic name used in the *createTopicDictionary* method, the name of the sensor *registerSensor* method and lastly in the way the Measurement object is created from the event in the *eventToMeasurement* method. See Figure 8.39 for code-snippets from the *eventToMeasurement* method in both the Magnetometer and Accelerometer modules.

Magnetometer

```
public ArrayList<BaseModel> eventToMeasurement(Event event) {
    ArrayList<BaseModel> measurements = new ArrayList<BaseModel>();
    float simulatedBearing = (float)event.getProperty(RobotConstants.MAGNETOMETER_EVENT_NAME);
    Measurement ms = new Measurement();
    ms.setSensorId(sensorID.get());
    ms.addLoad(createDataload(simulatedBearing));
    measurements.add(ms);
    return measurements;
}
private DataLoad createDataload(float bearing) {
    return new DataLoad(bearing, "bearing");
}
```

Accelerometer

```
public ArrayList<BaseModel> eventToMeasurement(Event e) {
    ArrayList<BaseModel> measurements = new ArrayList<BaseModel>();

    Acceleration simulatedAcceleration = (Acceleration)event.getProperty(RobotConstants.ACCELEROMETER_EVENT_NAME);
    Measurement ms = new Measurement();
    ms.setSensorId(sensorID.get());
    ms.addLoad(new DataLoad(simulatedAcceleration.getX(), "x"));
    ms.addLoad(new DataLoad(simulatedAcceleration.getY(), "y"));
    ms.addLoad(new DataLoad(simulatedAcceleration.getZ(), "z"));
    int boolInt = simulatedAcceleration.isBreaking() ? 1 : 0;
    ms.addLoad(new DataLoad((float)boolInt, "breaking"));
    measurements.add(ms);
    return measurements;
}
```

Figure 8.39: The two different versions of the *eventToMeasurement* method in the Magnetometer and Accelerometer modules.

Magnetometer and Accelerometer modules summarized

The two implemented Sensor Modules in the system, the Magnetometer and Accelerometer works by employing the *OSGiEventListener* block from the *Osgiblocks* module and the *SensorBlock* from the *General Blocks* module. The module listens for events published by the Simulator module under a Topic, translates events to measurements and stores them in the database.

In the next section the *PathFinder* module will be described. This module uses the output from the Sensors to calculate the movements the robot needs to complete, to reach its destination.

8.1.5 PathFinder module

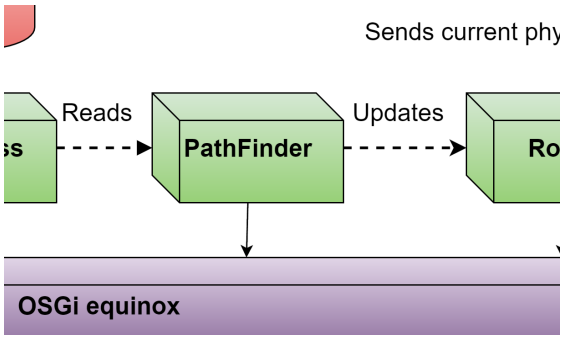

Figure 8.40: The PathFinder module

The *PathFinder* module was created to separate the control algorithm into a separate module. The module has its own life-cycle in the OSGi-framework, and as such has its own application block. The modules main responsibility is to extract sensor data from the database, use commands (destinations) received from an external source and combine this to calculate the *MovementState* the robot needs to be able to reach its destination. The combination of all MovementStates the robot enters from a position to a destination will in the end be the path the robot travels to reach its goal. The module contains the four following private blocks, used by the module to fulfil its purpose.

- Finder
- CommandHandler
- PathHandler
- PhysicalStateHandler

All blocks will be described in detail, the first block to be explained is the *application* block combining all the other blocks

PathFinder block

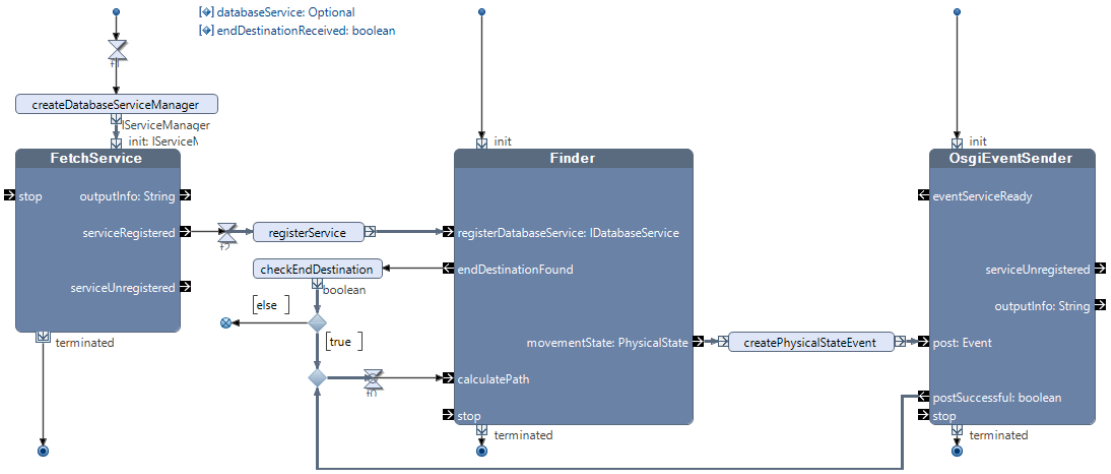

Figure 8.41: The PathFinder block

The Pathfinder block controls the life-cycle of the PathFinder and incorporates all of its private blocks. As can be seen in Figure 8.41, on the highest level the block uses the *FetchService* block to get access to the DatabaseService, it uses the OSGiEventSender to publish events through the OSGi framework and it uses the Finder block to calculate the needed movementstate to reach its destination. The databaseService found by the *FetchService* block is sent to the *Finder* block and used to extract sensor data from the database. The *Finder* module uses this data to calculate the movementstate and updated physical properties of the robot and sends this to the *OSGiEventSender* which in turn wraps it as an event and publishes it on the framework.

Finder block

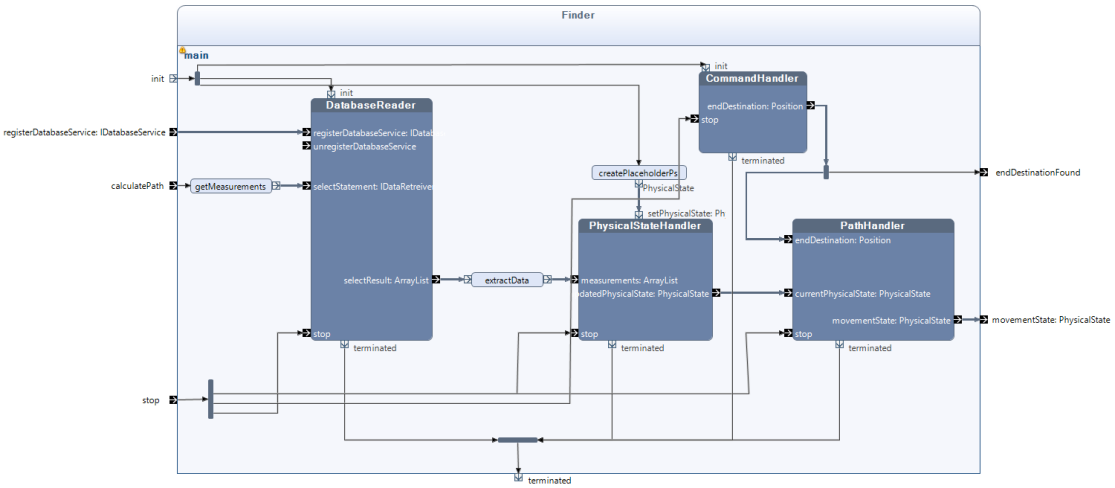

Figure 8.42: The Finder block

The *Finder* block was created to gather all the blocks directly involved in the control algorithm. The block is initiated by triggering the *init* input pin and enters the *waitingForDatabaseService* state. During initiation its inner blocks are started as well. These include the *DatabaseReader*, the *PhysicalStateHandler* and the *CommandHandler*.

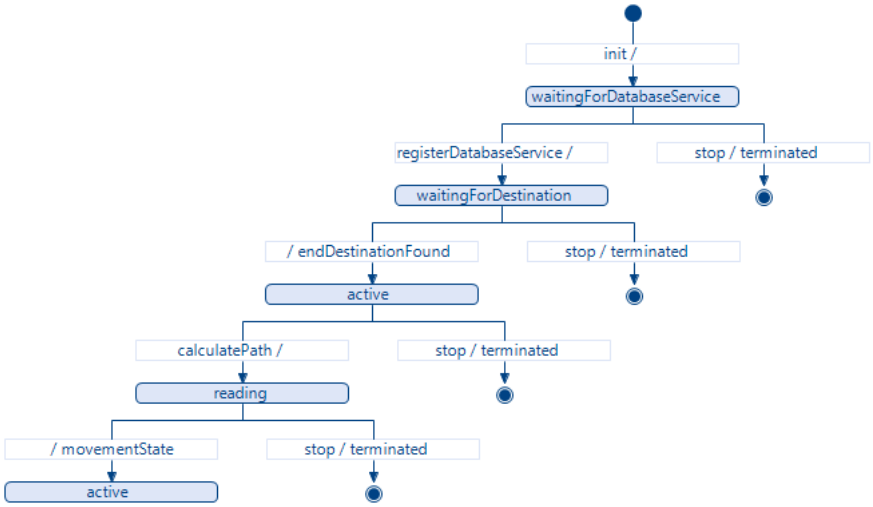

Figure 8.43: The Finder blocks external state machine diagram

The block changes state to *waitingForDestination* when it receives an DatabaseService object on the *registerDatabaseService* input pin. It stays in this state until the *CommandHandler* block receives a command containing a destination, this will trigger the transition into the state *active*. The block can now be used to retrieve data from the database by triggering the input pin *calculatePath*. A trigger on this pin calls the *getMeasurements* method which creates an object of the IDataRetriever object, sends this to the DatabaseReader which uses it to retrieve the most resent measurements from the database.

```
public IDataRetreiver getMeasurements() {
    return new IDataRetreiver(){
        @Override
        public ArrayList<BaseModel> createDataSetFrom(ResultSet resultSet) {
            ArrayList<BaseModel> result = new ArrayList<BaseModel>();
            Hashtable<Integer, BaseModel> tempRes = new Hashtable<Integer, BaseModel>();
            SimpleDateFormat sdf = new SimpleDateFormat("YYYY-MM-dd hh:mm:ss");
            try {
                while (resultSet.next()){
                    int measurementID = resultSet.getInt("mID");
                    int sensorID = resultSet.getInt("sID");
                    String tempDate = resultSet.getString("timestamp");
                    int dataLoadID = resultSet.getInt("dID");
                    int dataLoad_measurementID = resultSet.getInt("dmID");
                    float value = resultSet.getFloat("value");
                    String description = resultSet.getString("description");
                    String sensorName = resultSet.getString("sName");

                    java.util.Date dt = sdf.parse(tempDate);

                    Measurement ms = new Measurement(sensorID, dt, measurementID);

                    DataLoad dl = new DataLoad(dataLoadID, dataLoad_measurementID, value, description);
                    if (!tempRes.containsKey(ms.getSensorId())) {
                        result.add(new Sensor(sensorID, sensorName));
                        tempRes.put(ms.getSensorId(), ms);
                    }
                    ms = (Measurement) tempRes.get(ms.getSensorId());
                    if (ms.getId() == dl.getMeasurementId()) {
                        ms.addLoad(dl);
                    }
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
            tempRes.forEach((k,v) -> result.add(v));
            return result;
        }

        @Override
        public String getSelectStatement() {
            return "select m.id AS mID, "
                    + "m.sensor_id AS sID, "
                    + "s.name AS sName, "
                    + "m.timestamp AS timestamp, "
                    + "d.id AS dID, "
                    + "d.measurement_id AS dmID, "
                    + "d.value AS value, "
                    + "d.description AS description "
                    + "from measurement m, dataload d, sensor s "
                    + "where m.id = d.measurement_id "
                    + "and s.id = m.sensor_id "
                    + "ORDER BY timestamp DESC;";
        }
    };
```

Figure 8.44: Code-snippet showing the method *getMeasurements* creating an instance of the IDataRetriever interface to extract newly registered measurements from the database

Triggering the *calculatePath* method will transition the *Finder* block to the *read-*

ing state where it will wait for the result from the DatabaseReader. When the DatabaseReader gets the result from the database it outputs the result on the *selectResult* output pin, the result is sent through the *extractData* method and into the *PhysicalStateHandler* block where the measurements are used to update the physical state of the robot, in the next section, the *PhysicalStateHandler* will be described in detail. The updated physical state is then sent out on *updatedPhysicalState* pin and received by the *PathHandler* block. This block uses the physical state of the robot and the current destination to calculate the appropriate movement state needed to reach the destination, wraps the movement state in the Physical state object and sends it out on the *movementState* pin. When this is done the *Finder* block changes state back to the *active* state, awaiting a new trigger on the *calculatePath* input pin.

PhysicalStateHandler block

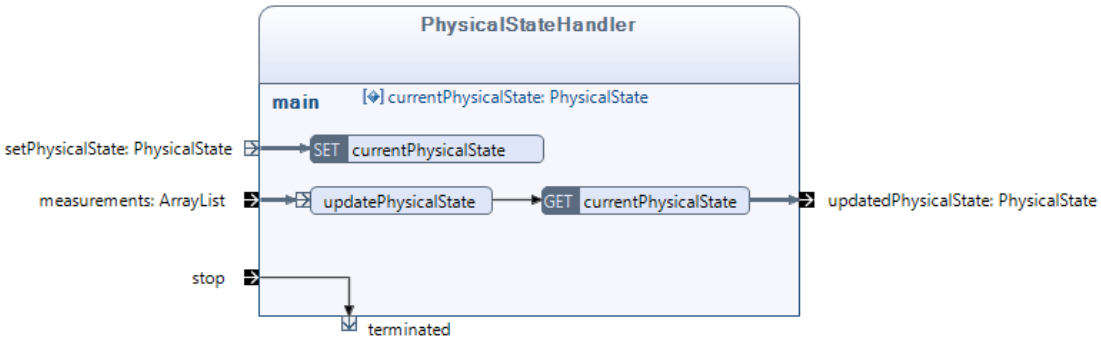

Figure 8.45: The PhysicalStateHandler block

The *PhysicalStateHandler* block has the responsibility of taking in the newly registered data from the sensors on the *measurements* input pin and use this to update the physical properties of the robot using the calculations and theory in section 7.3.3 *MVP Iteration 3: Simulate robot movement*.

CommandHandler block

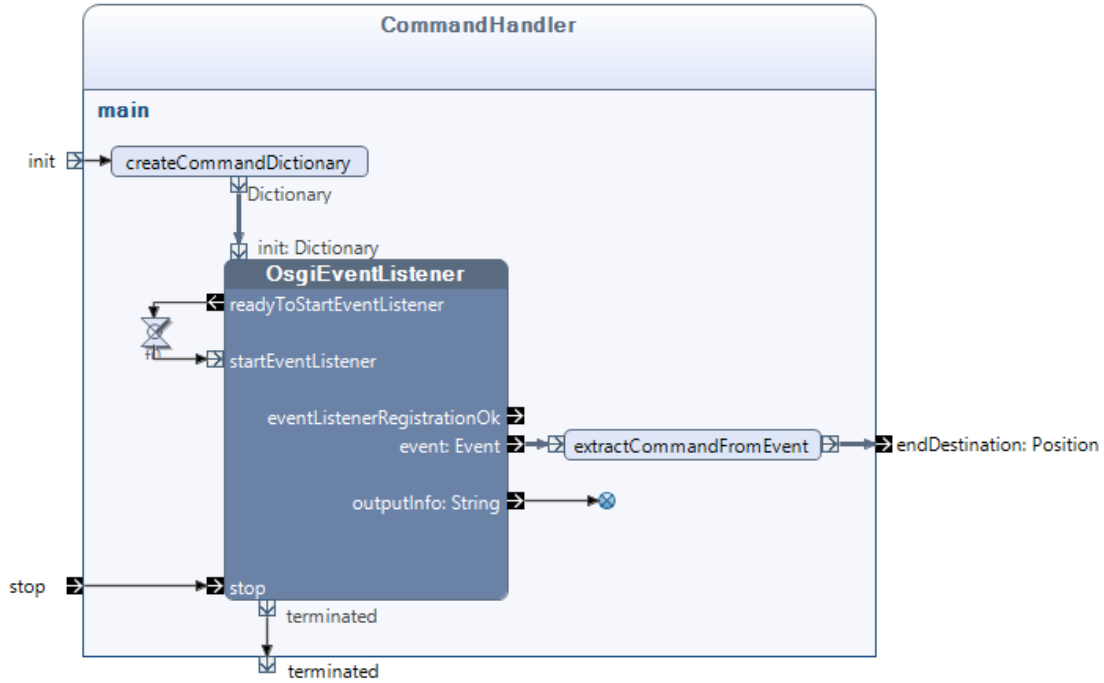

Figure 8.46: The CommandHandler block

To be able to calculate the path of the Robot, the PathFinder module needed a way to get the destination coordinates. The *CommandHandler* module was created to solve this issue. It uses the *OSGiEventlistener* block to listen for events published under the *Command* topic. These events contain an object of the *Position* class, see class diagram in Figure 8.17. When the *OSGiEventListener* gets an Command event on the *event* output pin it calls the *extractCommandEvent* method and send the contained destination out of the CommandHandler on the *endDestination* pin.

PathHandler block

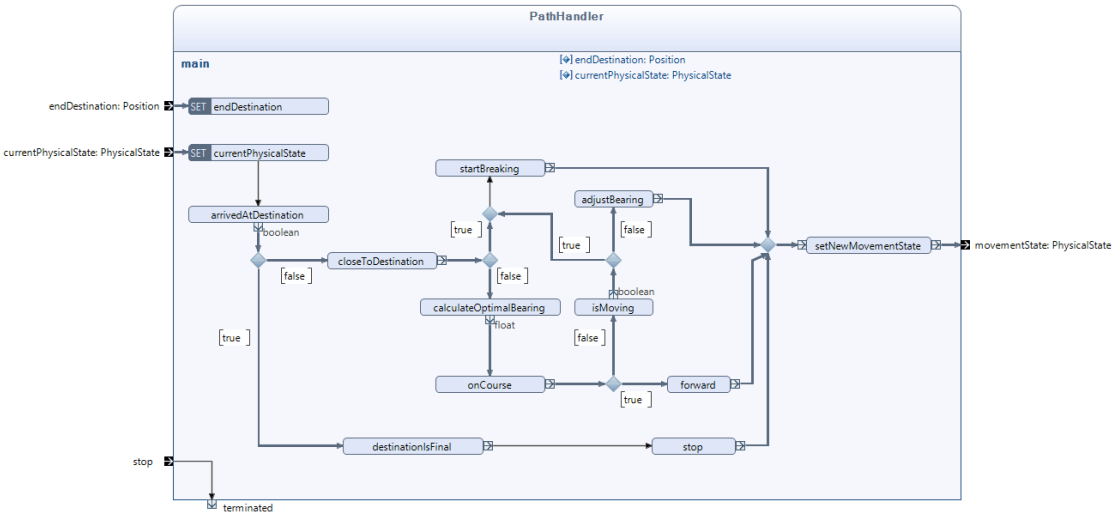

Figure 8.47: The PathHandler block

The *PathHandler* block contains the actual control algorithm created to satisfy *Minimum Viable Product iteration 4: Simulate the robot moving to a destination*. It has two input pins, *endDestination* and *currentPhysicalState* getting input from the *CommandHandler* and *PhysicalStateHandler* modules. The block is started by receiving a destination on the *endDestination* pin. When receiving an PhysicalState object, the control algorithm is started by traversing the methods shown in the block in Figure 8.47. The methods are the realisation of the activity diagrams in Figure 7.8 and Figure 7.9.

Based on the traversal of the PathHandlers methods, the block will run one of the following methods *startBreaking*, *adjustBearing*, *forward* or *stop*. These methods outputs MovementState instances that are sent to the *setNewMovementState* method which wraps the MovementState in the PhysicalState object and outputs the result on the *movementState* pin.

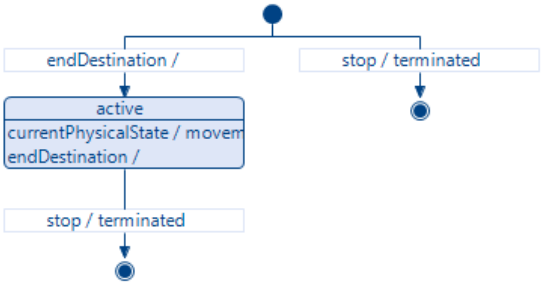

Figure 8.48: The PathHandler blocks external state machine

PathFinder module summarized

The PathFinder uses Sensor data registered by the sensors in the database combined with a destination and the current state of the robot to calculate which movement state the robot should enter to reach the destination. The main bulk of its work is done by the *Finder* block which in turn uses the *DatabaseReader* to retrieve data from the database, the *PhysicalStateHandler* to calculate the robots current physical properties, the *CommandHandler* to listen for destination commands and the *PathHandler* to calculate the next MovementState of the robot.

The PathFinder is the implementation of the equations and theory created to calculate the physical properties described in section 7.3.3 MVP Iteration 3: *Simulate Robot Movement*, and the control algorithm described in section 7.3.4 MVP Iteration 4: *Simulate the robot moving to a destination*

In the next section the *Robot* module will be described. The *Robot* module uses the output from the PathHandler to control the simulated engines.

8.1.6 Robot module

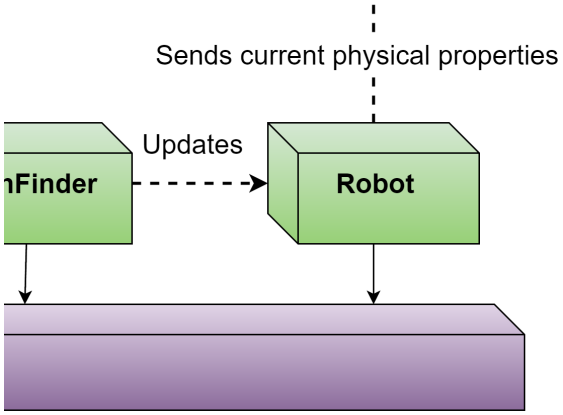

Figure 8.49: The Robot module

The Robot module was created to separate the path finding functionality from the functionality controlling the robots engines and thus making the system more modifiable.The Module is pretty simple, only containing one application block. The Robot module uses the MovementState created by the PathFinder to set the engines power output.

Robot block

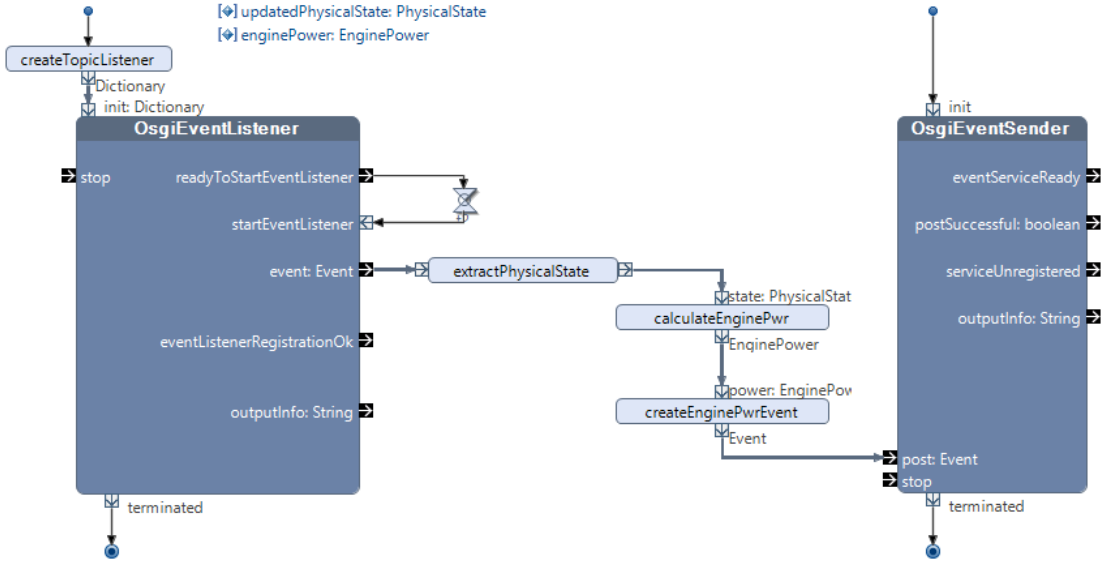

Figure 8.50: The Robot block

The robot block shown in Figure 8.50 is initiated by starting its two inner blocks; *OsgiEventListener* and *OsgiEventSender*. The OsgiEventListener starts listening for events published by the PathFinder containing the MovementState the robot needs to reach its destination. When an event is received on the *event* output pin, it is sent to the method *extractPhyiscalState* which extracts the PhysicalState object wrapped in the event. The MovementState stored in the PhysicalState object is then used by the *calculateEnginePwr* method to set the engine power to move the robot in accordance to the movement state, see Figure 8.51 for code-snippet.

```
public EnginePower calculateEnginePwr(PhysicalState physicalState) {
    switch (physicalState.getMovementState()) {
    case FORWARD:
        return new EnginePower(100, 100);
    case BREAKING:
        return new EnginePower(0, 0);
    case ROTATE_LEFT:
        return new EnginePower(-100, 100);
    case ROTATE_RIGHT:
        return new EnginePower(100, -100);
    default:
        return new EnginePower(0, 0);
    }
}
```

Figure 8.51: The *calculateEnginePwr* method

The EnginePower object is wrapped in an Event object in the *createEnginePwrEvent* and sent to the *OsgiEventSender* block which publishes the event on the OSGi framework.

In the next section the *Simulator* module will be described. This module listens for both the *EnginePower* object published by the Robot module and the *PhysicalState* object published by the PathFinder module.

8.1.7 Simulator module

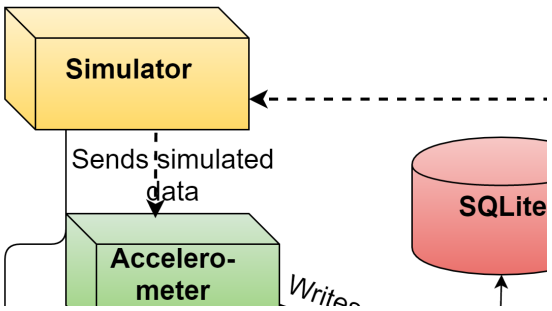

Figure 8.52: The Simulator module

The *Simulator* module, the last piece of the Simulation loop was created to confine all the code and functionality related to the simulation into one module. The *Simulator* module exports no classes nor any blocks, and only contain one private block, the *Simulator* block.

Simulator application block

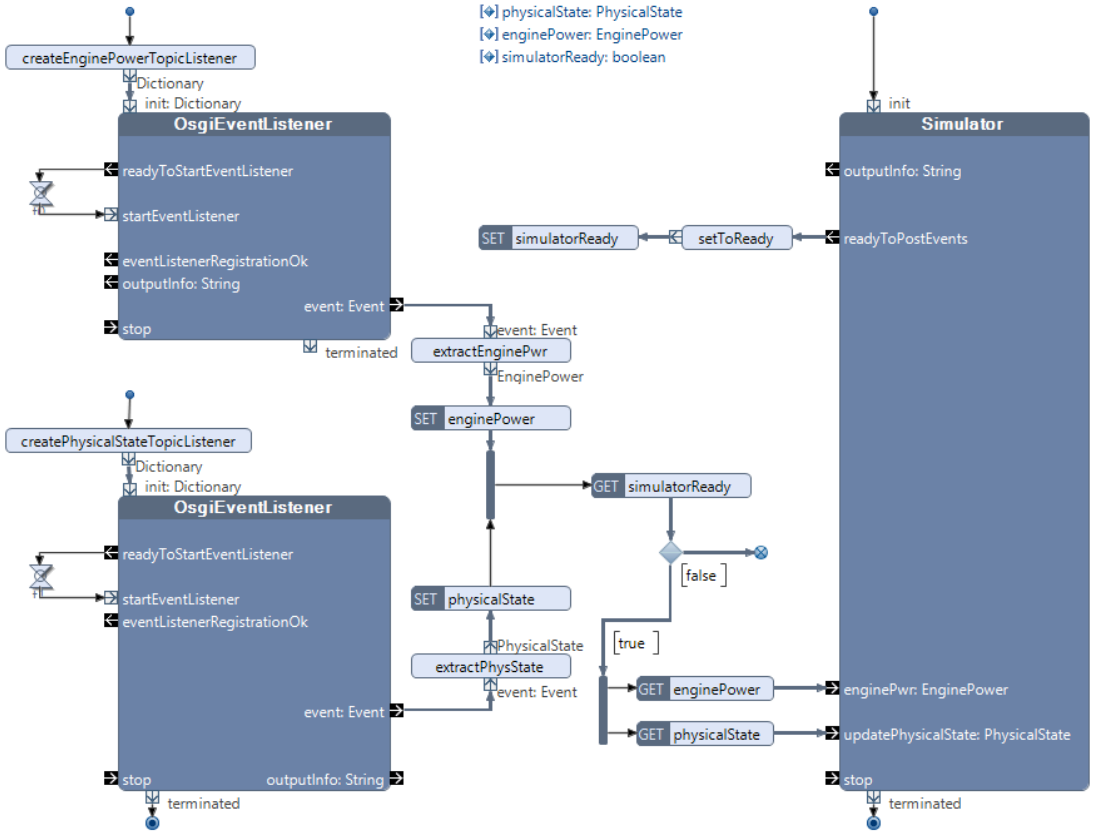

Figure 8.53: The Simulator application block

The *Simulator* application block consists of three inner blocks, the *Simulator* building block and two *OsgiEventListener* blocks. The two OsgiEventListeners are started with the creation of topic dictionaries, there the top most one in Figure 8.53 listens for events published under the EnginePower topic and the other listens for events published under the PhysicalState topic.

When receiving events, the Simulator *application* block extracts the data from the events and store the data in the public fields *physicalState* and *enginePower*. The next step is to check whether the Simulator *building* block has been started, if it has not the events are discarded, if it has been started the data from the enginePower and physicalState fields are sent to the Simulator *building* block.

All the logic and functionality related to creating the simulated data from the physical properties and the engine power was placed in its own block, the Simulator

building block.

The next section will describe the functionality of the Simulator *building* block seen on the right side of Figure 8.53

Simulator building block

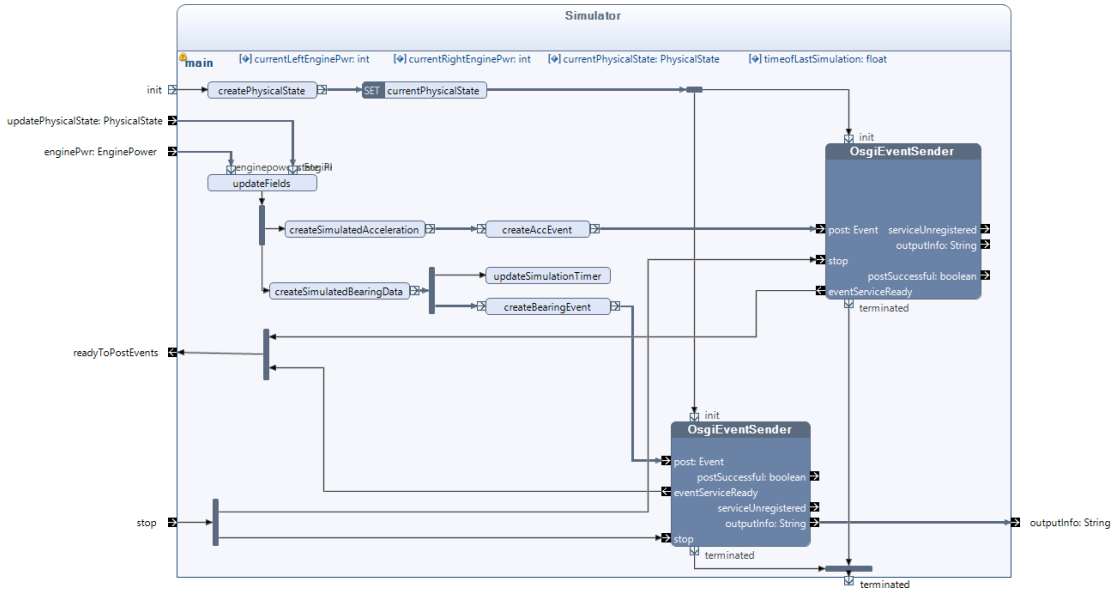

Figure 8.54: The Simulator building block

The Simulator block is started by triggering the *init* input pin, calling the method *createPhysicalState* and then initiating the two *OsgiEventSender* blocks. The createPhysicalState method is used to create a placeholder object used until the first PhysicalState object is received on the *updatePhysicalState* input pin. During initiation the block enters the *waitingForEventSenders* state and stays here until the *OsgiEventSenders* has completed their start-up process. The OsgiEventSenders are dependent on getting access to the *Event Admin service*, and will trigger the outgoing *eventServiceReady* pin when completed. When both blocks are ready, the Simulators *readyToPostEvents* output is triggered and the block enters the *active* state, see exteram state machine in Figure 8.55.

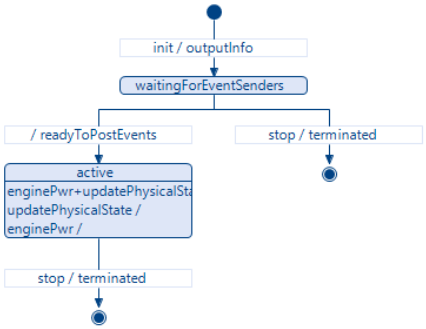

Figure 8.55: External state machine of the Simulator building block

While the block exist in the *active* state it will accept input on the two input pins *updatePhysicalState* and *enginePwr*. The input will be sent to the *updateFields* method, updating the global fields with new data and then calling the two methods *createSimulatedAcceleration* and *createSimulatedBearingData* in parallel.

The createSimulatedAcceleration sets the acceleration in accordance to the MovementState of the robot. In the case of the current control algorithm limited by the simplifications listed in section 7.2, the acceleration for x direction is set to maximum in the forward movement state until it reaches maximum speed, or maximum deceleration in x-direction if the robot is breaking until it has stopped. In rotation state it sets the acceleration to zero, see Figure 8.56 for code-snippet from the implementation.

```
public Acceleration createSimulatedAcceleration() {
    Acceleration acceleration;
    float xAcc;
    float yAcc;
    float zAcc;
    boolean breaking;
    switch(currentPhysicalState.getMovementState()) {
        case FORWARD:
            if (currentPhysicalState.getVelocity() >= MAX_SPEED) {
                xAcc = 0f;
                yAcc = 0f;
                zAcc = 0f;
            } else {

                xAcc = Acceleration.getXAcceleration(RobotConstants.MAX_ACCELERATION_PR_SECOND, 0);
                yAcc = Acceleration.getYAcceleration(RobotConstants.MAX_ACCELERATION_PR_SECOND, 0);
                zAcc = 0f;
            }
            breaking = false;
            break;
        case BREAKING:
            xAcc = Acceleration.getXAcceleration(RobotConstants.MAX_DECELERATION_PR_SECOND, 0);
            yAcc = Acceleration.getYAcceleration(RobotConstants.MAX_DECELERATION_PR_SECOND, 0);
            zAcc = 0f;
            breaking = true;
            break;
        default:
            xAcc =  0f;
            yAcc =  0f;
            zAcc =  0f;
            breaking = false;
            break;
    }
    acceleration = new Acceleration(xAcc, yAcc, zAcc, breaking);
    return acceleration;
}
```

Figure 8.56: Code-snippet showing the createSimulatedAcceleration method

To create simulated bearing data the methods shown in Figure 8.57 is used. The *createSimulatedBearingData* method starts by calling the *calculateDeltaBearing* method to calculate the amount of rotation that should be added to the current bearing based on the time since the last time it was calculated and stores this value in the deltaBearing variable. After this has been done the method checks whether the robot is rotating to the relative left or right and either adds to or subtracts the deltaBearing value from the current bearing.

```
public float createSimulatedBearingData() {
    float deltaBearing = calculateDeltaBearing();
    switch (currentPhysicalState.getMovementState()) {
        case ROTATE_LEFT:
            return validBearingValue(currentPhysicalState.getBearing() + deltaBearing);
        case ROTATE_RIGHT:
            return validBearingValue(currentPhysicalState.getBearing() - deltaBearing);
        default:
        return currentPhysicalState.getBearing();
    }
}

private float calculateDeltaBearing(){
    float deltaTime;
    if (timeOfLastSimulation == 0) {
        deltaTime = 0f;
    }
    else {
        deltaTime = System.nanoTime() - timeOfLastSimulation;
    }
    float deltaTimeSec = deltaTime/1000000000f;
    return deltaTimeSec * RobotConstants.MAX_DEGREES_PR_SECOND;

}

private float validBearingValue(float bearing) {
    if (bearing >= 0 && bearing <= 359) {
        return bearing;
    }
    if (bearing > 359) {
        return bearing % 359;
    }
    return 360 - bearing;
}
```

Figure 8.57: Code-snippet showing the methods used to create simulated bearing data

After the simulated data has been created the *Simulator* block wraps the data in Event objects under simulation data topics, one for each sensor, and uses the *OsgiEventSenders* to publish the data on the OSGi framework. These are the events subscribed to by the two sensors modules Accelerometer and Magnetometer, described in section 8.1.4.

Simulator module summarized

The *Simulator* module subscribes to data topics delivered by the *PathFinder* and the *Robot* module. It subscribes to the Pathfinders *physical state* topic to get the current physical properties of the robot and to the Robot modules *engine power* topic to get the current engine power output. The module uses the data from

these modules to simulate the change to the current acceleration and bearing of the data, and publishes the newly updated values as events on the framework. The Magnetometer and Acceleration modules get their data from subscribing to these events.

In the next section the last module in the system, the *Control Panel* module will be described. The module uses the physical properties published by the Pathfinder to draw a graphical simulation of the robot moving towards the destination.

8.1.8 Control Panel module

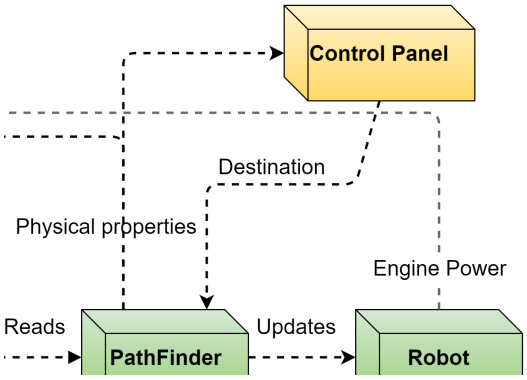

Figure 8.58: The Control Panel module

The *Control Panel* is the only module with a life-cycle in the OSGi framework that is not a part of the Simulation Loop. The control panel module was created to satisfy the fifth minimum viable product iteration, *Graphical Simulation*. The module is also used as a way for the user of the simulation software to input a destination for the robot, this way the destination can be changed run-time. The module does not export any classes or blocks to other modules in the system and does not register any services in the OSGi framework.

The next section will describe the implementation of the module in Reactive Blocks.

Control Panel block

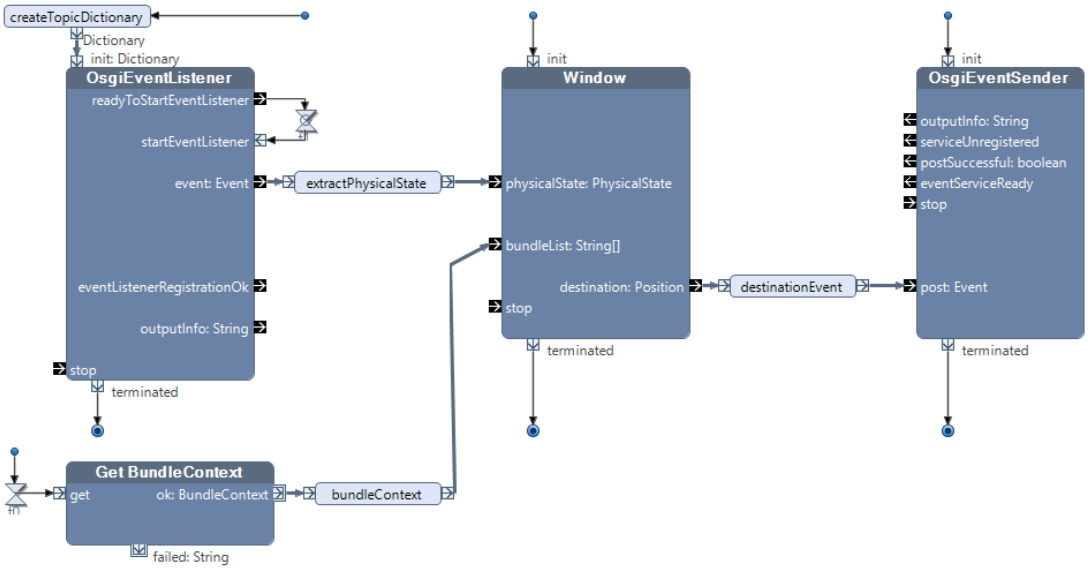

Figure 8.59: The Control Panel implementation in Reactive Blocks

The Control Panel application block controls the life-cycle of the Control Panel module and consists of the blocks shown in Figure 8.59. The block use an *OsgiEventListener* block to listen for PhysicalState objects published by the *PathFinder* module, an *OsgiEventSender* to publish destination events used by the *PathFinder* module, the *Get BundleContext* block to get access to the bundleContext and lastly its private block *Window* to draw the Graphical User Interface.

Most of the Control Panel's core functionalities reside in the *Window* block which receives the PhysicalState object every time the PathFinder module publishes a new update of the robots Physical Properties. How it uses this data and the bundle context will be explained in the next section.

Window block

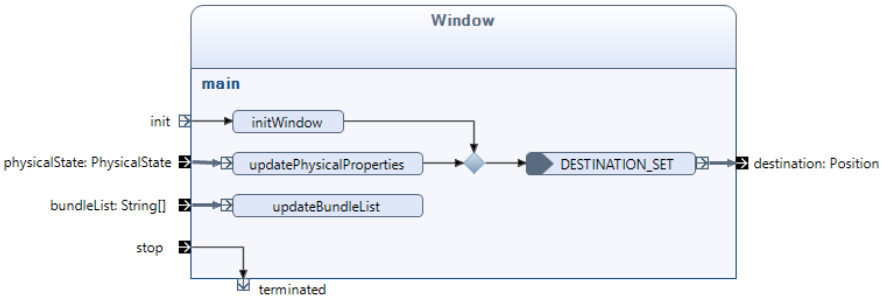

Figure 8.60: The Window block

The Window block is initiated by triggering the *init* input pin. This will call the method *initWindow* which draws the Graphical User Interface, and start the DESTINATION_SET event listener. The GUI is created by a class called ControlWindow containing and controlling all the GUI components, see Figure 8.60 for screen shot of the GUI.

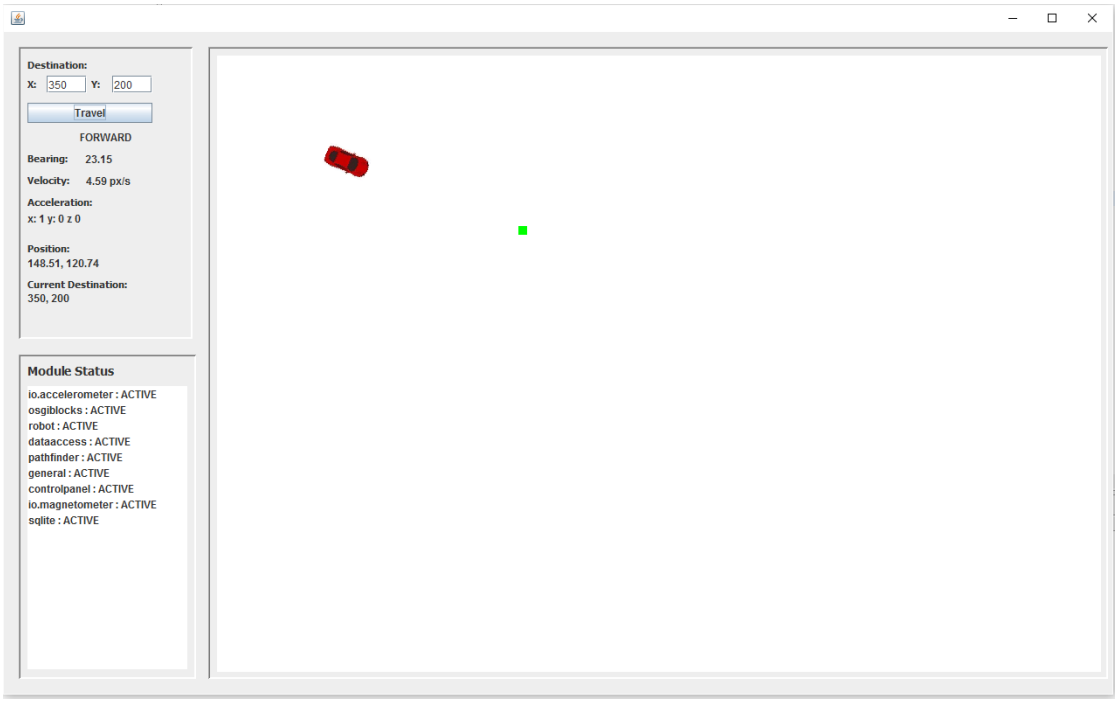

Figure 8.61: Screen-shot of the GUI.

Entering a destination in the input fields shown on the top-left in the GUI, and pressing the button *Travel* triggers the DESTINATION_EVENT, containing the destination, to be sent to the *Window* block. The Position object containing the destination is sent out of the block on the outgoing *destination* pin.

Every time the Window block receives an PhysicalState object on the *physicalState* input pin, it updates the fields seen on the top-left of the GUI screen-shot in Figure 8.60. The canvas used to draw the simulated robot (red car) and the destination(green square) is run on an own Thread by using the class *CustomCanvas*. The *CustomCanvas* implements the Runnable interface and extends the Canvas class. In Figure 8.63 the CustomCanvas loop can be seen calling the *render* method every 17th millisecond to achieve 60 frames per second.

```
@Override
public void run() {
    while(running){
        render();
        try{
            Thread.sleep(17);
        } catch(Exception e){
        }
    }
}
```

Figure 8.62: Code-snippet showing the CustomCanvas loop calling the *render* method every 17th millisecond

The render method renders the canvas every 17th millisecond, and can be seen in Figure 8.63. It uses a buffer strategy to double buffer the canvas reducing lag and calls the *draw* method to draw the robot and the destination onto the canvas. The draw methods can be seen in figure 8.63

```
public void render(){
    BufferStrategy bs = getBufferStrategy();
    this.invalidate();
    if(bs == null){
        createBufferStrategy(2);
        return;
    }
    Graphics g = bs.getDrawGraphics();
    g.setColor(Color.WHITE);
    g.fillRect(0, 0, getWidth(), getHeight());
    draw(g);
    g.dispose();
    bs.show();
}

public void draw(Graphics g){
    drawDestination(g);
    drawRobot(g);
}
public void drawRobot(Graphics g) {
    if(physicalState == null)
        return;
    Position robPos = physicalState.getPos();
    if(robotImg != null){
        Graphics2D g2d = (Graphics2D) g;
        double imgW = robotImg.getWidth()/2;
        double imgH = robotImg.getHeight()/2;
        double robX = robPos.getxPos() - imgW;//center image on anchor
        double robY = robPos.getyPos() - imgH;//center image on anchor
        AffineTransform af;
        af = AffineTransform.getTranslateInstance(robX, robY);
        double bearingInRadians = Math.toRadians(physicalState.getBearing());
        af.rotate(bearingInRadians, imgW, imgH);//rotate
        g2d.drawImage(robotImg, af,null);
    }else {
        g.setColor(Color.black); //if robot image unavailable, use a black square
        g.fillRect((int)(robPos.getxPos()-5), (int)(robPos.getyPos()-5), 10, 10);
    }
}

private void drawDestination(Graphics g) {
    if(destination == null)
        return;
    g.setColor(Color.green);
    g.fillRect((int)(destination.getxPos()-5), (int)(destination.getyPos()-5), 10, 10);
}
```

Figure 8.63: The methods used to draw the robot and the destination on the Canvas

The *Window* block can continuously receive input on the *bundleList* input pin, the list is then rendered on the GUI, see the lower left panel in Figure 8.61. The list contains all bundles related to the system registered in the OSGi framework, as well as their status.

Control Panel module summarized

The Control Panel subscribes to the PhysicalState topic published by the *PathFinder* to get the robots current physical properties. The module uses the *Window* block to draw a simulated graphical robot on the screen and to list all the physical properties of the robot. The GUI created by the *Window* gives the user the option to enter a destination for the robot, which is published under the command topic. The *CommandHandler* block in the *PathFinder* module subscribes to this topic and uses the destination to set the current movement state of the robot, see section 8.1.5 PathFinder module. The module also lists all modules related to the system in the GUI, including their current status.

8.2 Prototype implementation

During the development and implementation of the 5 minimum viable product iterations the system gradually evolved into the prototype system described in section 8.1, System Design. The prototype completes the following five minimum viable product iterations:

- Inter-modular Communication
- Simulated sensor modules
- Simulate robot movement
- Simulate robot moving to a destination
- Graphical simulation

8.2.1 Running the prototype

Through Figures 8.64 to 8.70 screen-shots taken from a live simulation of the robot can be seen. The robots initial position was set to the coordinates 100, 100 and its destination was set to 250, 300, see Figure 8.64. The destination error margin is set to 10 pixels, e.g. the robot will accept a destination with an error margin of 10 pixels in both x and y directions. The bearing error margin is set to 5 degrees, this means that the robot accepts an bearing deviating 5 degrees from the optimal bearing when moving towards the target.

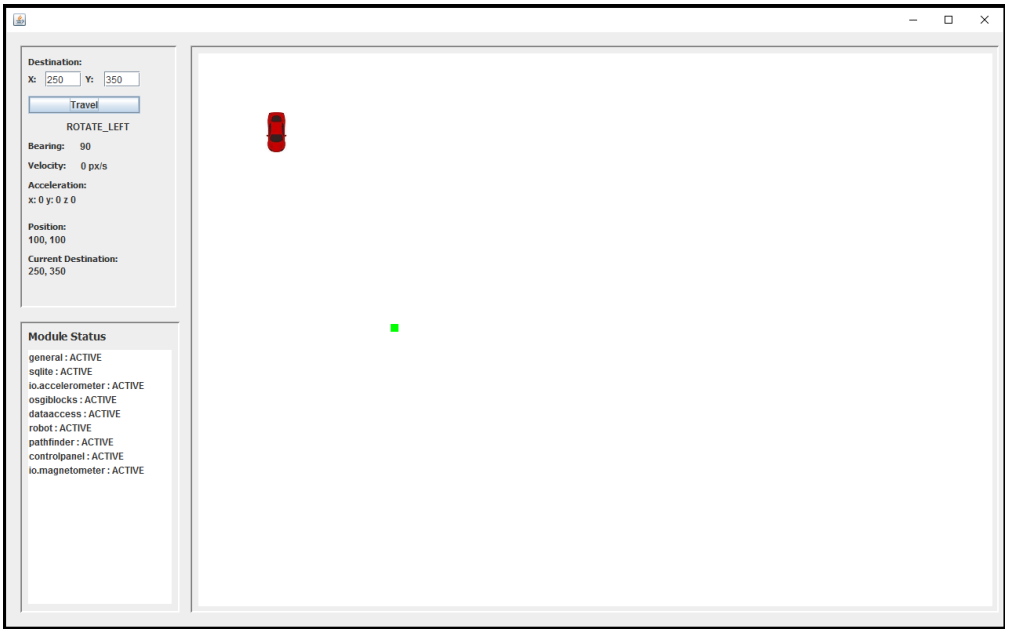

Figure 8.64: First screen-shot of live simulation, start position of the robot.

The robot starts in its initial position 100, 100 and with a bearing of 90 degrees. The Destination has just been set to 250, 350 and the robot has entered the ROTATE_LEFT state to adjust its bearing towards the destination.

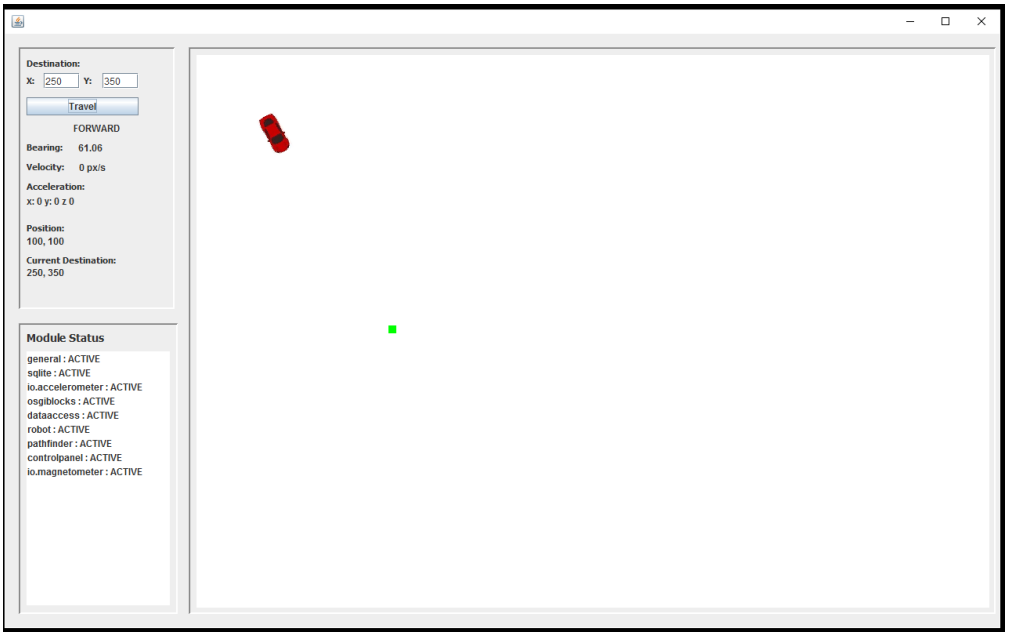

Figure 8.65: Secon screenshot of live simulation, robot starts moving towards target

The robot is finished adjusting its bearing, which is now at 61.06 degrees and has changed state to FORWARD ready to move towards the destination.

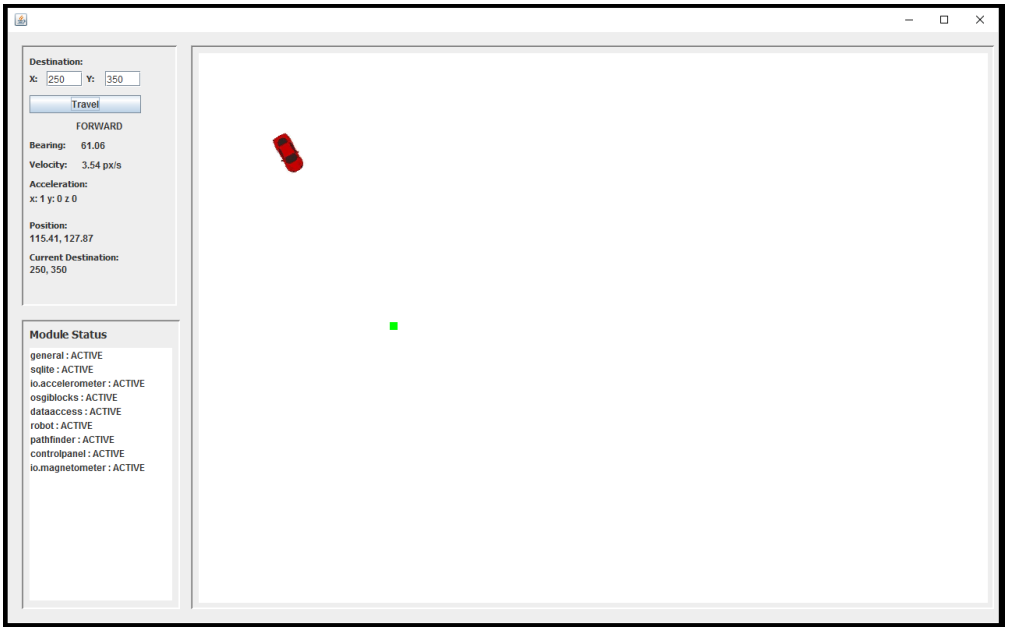

Figure 8.66: Third screenshot from live simulation, the robot is building speed towards its destination, accelerating with 1 px/s^2

The robot has been moving forwards a while, with an 1 px/s^2 acceleration in the x-direction and has built a velocity of 3.54 px/s. The position of the robot is now 115.41, 127.87.

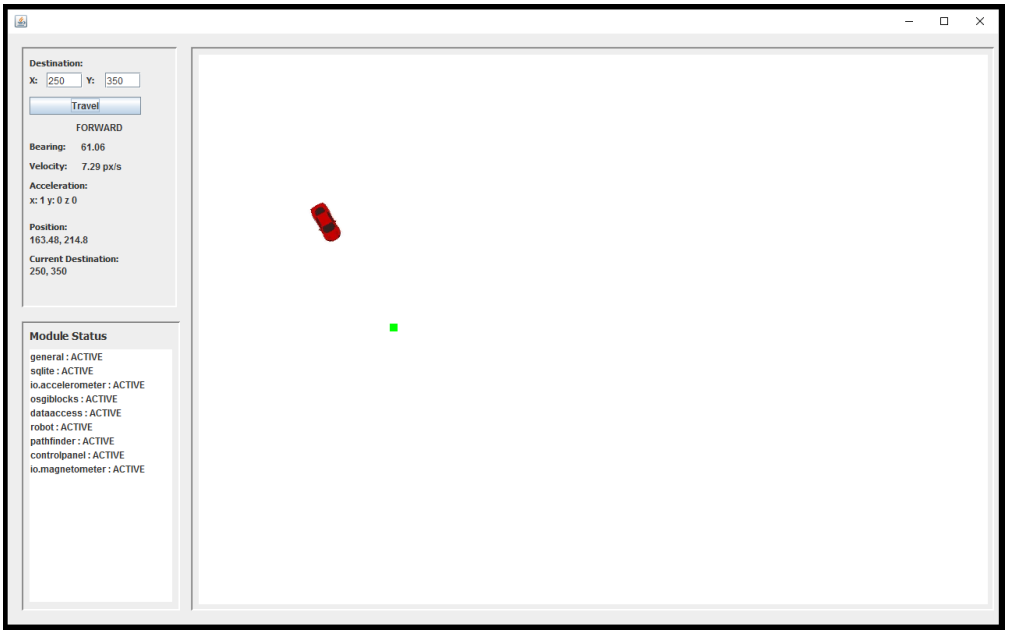

Figure 8.67: Fourth screen-shot from live simulation, robot still accelerating towards destination

The robot is still accelerating towards the destination, now with an velocity of 7.29 px/s and has now reached the position 163.48, 214.8.

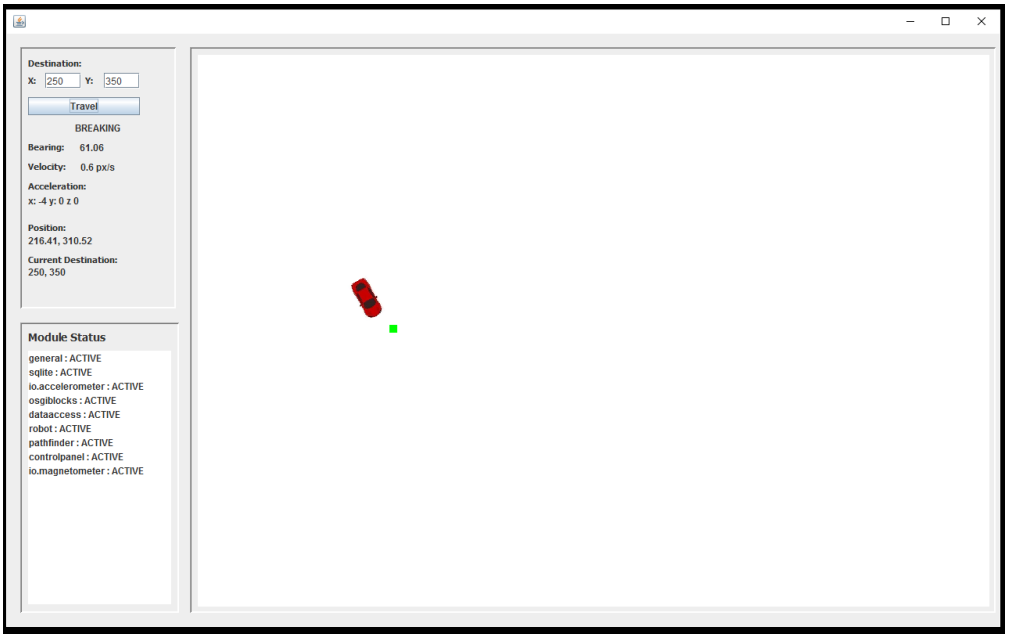

Figure 8.68: Fifth screen-shot from live simulation, the robot is breaking to adjust its bearing

The closer the robot comes to its destination, the higher the initial bearing error becomes. In Figure 9.1 the robots bearing deviates more than 5 degrees from the optimal bearing, and is breaking to be able to enter the ROTATE_LEFT state. The velocity is almost at 0 px/s, which is required for the robot to start rotating.

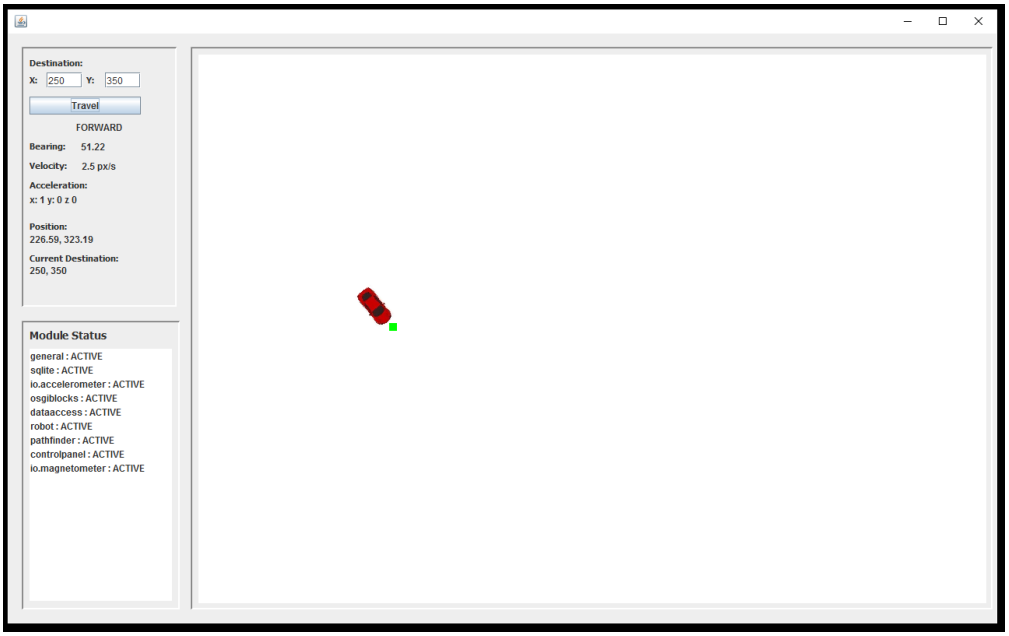

Figure 8.69: Sixth screen-shot from the live simulation, the robot has adjusted its bearing and is moving towards the destination

The robot is now finished rotating, has a current Bearing of 51.22 degrees and is accelerating towards the destination.

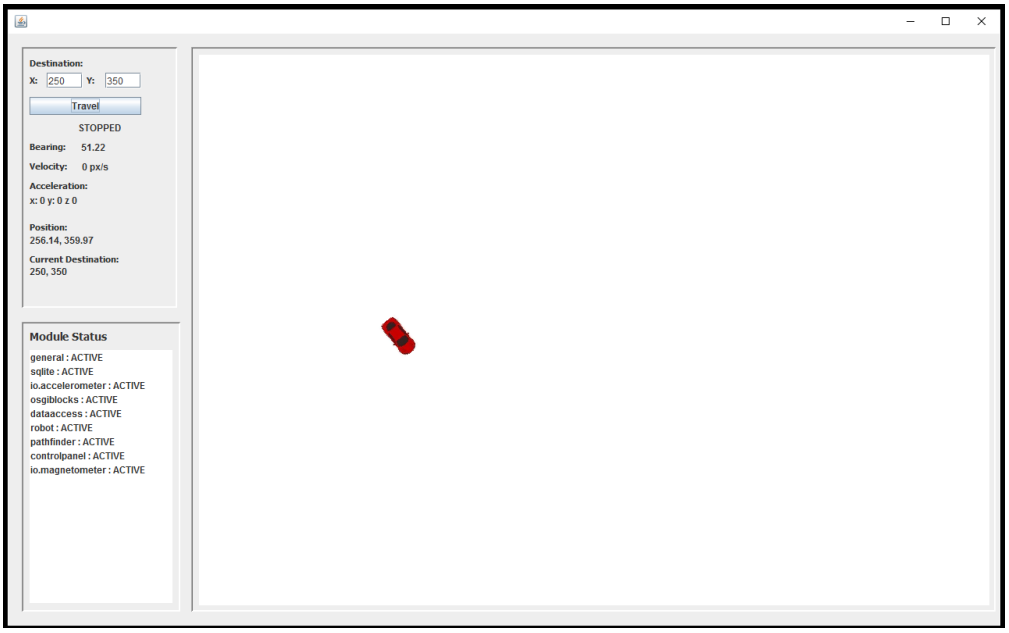

Figure 8.70: The last screen-shot of the live simulation, the robot reached the destination

The robot has reached its destination, and entered the STOPPED state.

Part IV

Summary

Chapter 9

Discussion

Before the system design and implementation was initiated, the following research questions were defined:

- What kind of software architecture is available for a highly modularised system for controlling transport robots?
- Are there any proof of concept systems, or control software systems created for robots using the OSGi framework?
- How much of the work conducted in my specialization project can be used in this master thesis?

Based of the research done to answer these questions the following Minimum Viable Product Iteration were created:

- MVP Iteration 1: Inter-modular Communication
- MVP Iteration 2: Simulated sensor modules
- MVP Iteration 3: Simulate robot movement
- MVP Iteration 4: Simulate robot moving to a destination
- MVP Iteration 5: Graphical simulation

The relevance of the research questions will be discussed in section 9.1 and the realization, implementation and usefulness of the MVP iterations will be discussed in section 9.2.

9.1 Process

The description for this master thesis is *Scalable Self-Adaption Control system for simulated transport robots*. The goal was to design a highly modularized system to control simulated transport robots. In addition to the design, the tool *Reactive Blocks* (2) and the OSGi framework was going to be used to create a prototype of the system.

In the start of the project a set of research questions were created to narrow the literature research and preparations needed before creating the system design, see section 2.2. The creation of the questions was of great help by filtering out unnecessary research and focus my mind on the correct path. Most articles I read, were selected to answer the research questions and most were of relevance to the project.

Based on the work done during the literature research, five Minimum Viable Product iterations were defined, see section 2.3.1. The theory and technology needed for each minimum viable product iteration was researched and documented, see section 7.3. Using the Minimum Viable Product technique described in section 2.3.1 helped channeling the work into the most important aspects of the system at all times. The technique helps focusing the mind by limiting the amount of functionalities being developed at the time. During the development of one iteration, the focus could solely be on getting the functionalities in that iteration to work. Since the development of the system included several unknown factors with a lot of uncertainties e.g. combining OSGi and Reactive Blocks, developing robotics system using OSGi, combining all this with an SQLite database, getting the inter-modular communication to work properly and so on, the use of the MVP technique provided the shortest way to a functional system and made sure that there always was a system to fall back on, with at least some of the systems core functionalities in place.

9.2 System Design and implementation

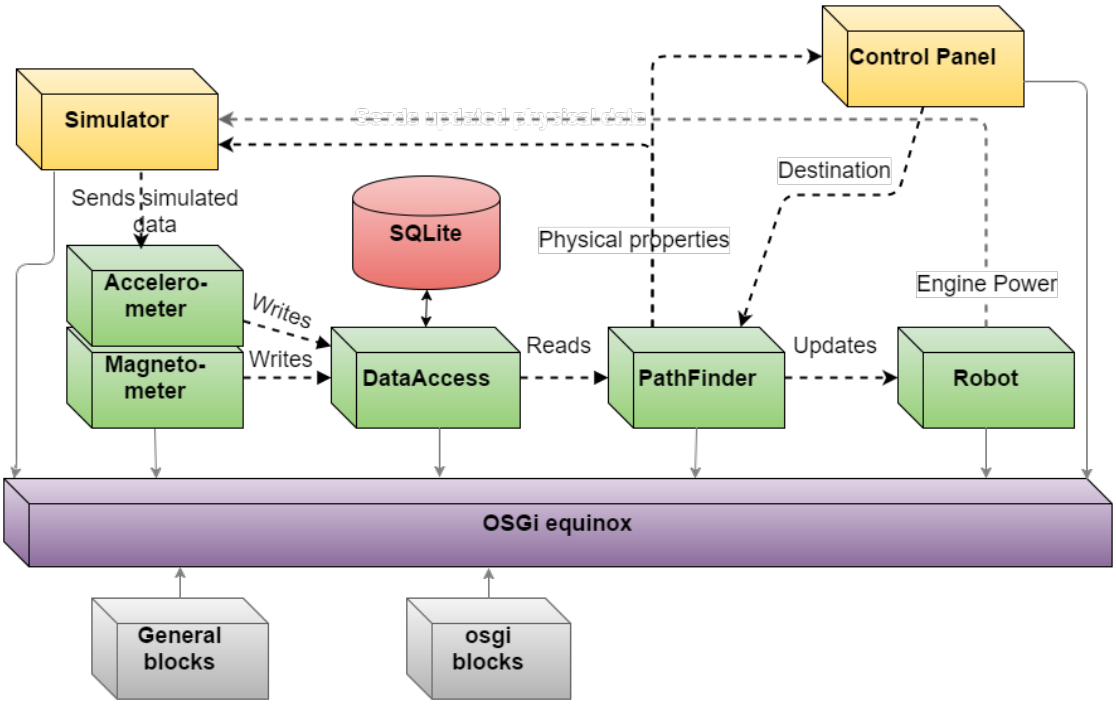

Figure 9.1: Final version of the system design and all its implemented modules

The goal of the project was to design a highly modularized control system for simulated transport robots. The system needed modules with high cohesion and low coupling to support the modules with separate life-cycles, able to be independently installed, uninstalled, updated and stopped. The final system design and the design of the implemented prototype was described in section 8.1, and it will now be discussed if the system design achieved its goals and if the implementation of the modules achieved the goals of the minimum viable product iterations.

9.2.1 System architecture and design

To achieve a highly modularized system there were two key factors that had to be achieved in the design and the implementation of the prototype, high cohesion and low coupling.

High Cohesion

High cohesion is the collection of similar elements of the system into separate modules. The contents of a module should reflect the name and the responsibility of the module, and should not contain anything else. It is also important that no elements related to the modules responsibility is placed anywhere else.

This was the foundation of the current systems design, and the implemented modules reflect this work. The current design and implementation achieves this, in my opinion. All modules in the system have a high responsibility density, and little to none "bleeding" of their responsibility domain into other modules.

Low Coupling

Its not enough to have high cohesion if the modules are highly dependent and closely coupled to others. The modules should be able to communicate, but with as little as possible connection and knowledge of the other modules. The current implementation of a database and publish-subscribe pattern achieves, in my opinion, just that. They only share the knowledge of topics to publish under and subscribe to, but not from where it originated. It is also flexible in the way that it is possible to add subscribers to a topic without having to change the current system or notify any of the existing modules.

Modifiability in practice

Other than implementing the current system modules and working through the minimum viable product iterations, there was little to none testing done on how easy it was to add/remove modules of the system. The only real test of the systems flexibility and ease of modifiability was done when adding the *Control Panel* module to the already functional simulation system. This was in my opinion done very fast, and with little to none changes necessary to the rest of the system. The work on the module was pretty much only related to the modules use-case and not to changing the current system to incorporate the new module.

The next few sections will discuss how the minimum viable product iterations were achieved when creating the system modules.

9.2.2 Realization of MVP 1: Inter-modular Communication

When creating a highly modularized system with independent modules, the communication and cooperation between the modules become an issue. To solve this a database for the communication between the Sensor and Pathfinder modules were used, and an implementation of the publish-subscribe pattern as communication between the rest, see section 7.3.1. The goal was also to create these communication facilities in a way that hides its complexity and provide an easy-to-use interface for the modules. The publish-subscribe pattern was realized with the *OsgiEventSender* and *OsgiEventListener* blocks exported by the *Osgiblocks* module. These blocks can be used by the modules in the system to communicate, providing a means of subscribing to and publishing events through the OSGi framework. The *DataAccess* module with its *DatabaseService* was created to give modules the ability to read and write data to and from the database, by using the databaseService registered in the OSGi framework. To access and register services in the OSGi framework the *RegisterService* and *FetchService* blocks were added to the osgiblocks module.

The design and implementation of the inter-modular communication was imperative to the control software. Without a loosely coupled way for the modules to communicate together, there was no way to reach the goal of a highly modularized system.

9.2.3 Realization of MVP 2: Simulated sensor modules

Another corner stone of automated control software is the sensors. Without sensors there is no way for a real or simulated robot to navigate. In the simulation software the functionality of the sensor modules are limited, their responsibility is to receive data from some unknown source, in the simulation software that unknown source is the *Simulator* module, and store this data in the database. The database technology selected for the simulation software was a SQLite database such that the *DataAccess* module could have full control over the creation, maintenance, communication and life of the database. The problem with the *SQLite* database was that it does not support concurrent writing, in the control system the Sensor modules will register data concurrently. This was solved by creating a read/write Thread for the DatabaseService, see section 8.1.3. The sensors registers their measurements in a buffer, which the databaseService pulls at given intervals and writes to the database.

In practice the DataAccess and Sensor modules worked pretty well and the use of an SQLite database was a good decision. One can argue that using a database

with the option of concurrent writing might have been a better solution, and that it would increase the rate of with sensors could write to the database. With concurrent writing one could allow for different write speeds for different sensors based of how important their input was. The prototype of the simulation system was set to write the buffer to the database every 200 milliseconds, this may have been to slow for certain sensors, for example laser sensors for measuring distance to potential collisions with other fast moving objects. The alternative to using another database technology might have been to create a direct publish-subscribe path for these kind of sensors from the Sensor Module to the *PathFinder* module, circumnavigating the DataAccess module.

9.2.4 Realization of MVP 3: Simulate robot movement

Simulating the movement of the robot was the first real proof of concept for the already implemented minimum viable product iterations. The early stages of robot movement was achieved by connecting the Simulator module, the Sensor modules and an early version of the PathFinder module, where the only block implemented was the PhysicalStateHandler, see section 7.3.3 for the calculations done by the block and Figure 8.45 for the implementation. The simulator module provided the Sensor modules with data which they stored in the database, the PathFinder read the data from the database and sent the updated physical properties directly to the Simulator module. The simulator module used this information to send new data to the Sensor modules and so on. The Physical properties were printed to the console for each iteration.

The completion of this minimum viable product iteration provided a proof of concept for the early stages of the Simulation loop, and showed that it was possible to use both the DatabaseService and the publish-subscribe functionalities to communicate between the modules.

9.2.5 Realization of MVP 4: Simulate the robot moving to a destination

The goal for an autonomous robot is to be able to handle some kind of predefined task without human interference. The goal of this specific simulated robot was to use simulated data from an Magnetometer and Accelerometer to reach a destination. The fourth minimum viable product(MVP) iteration was to create and implement the necessary code to achieve this. In the third MVP iteration the PathFinder module was created, with the functionality of changing the physical

properties of the robot for each iteration of the simulation loop. In this MVP iteration the PathFinder was expanded to include the *Finder* block and all its internal blocks except the *CommandHandler*, see section 8.1.5, which contained the algorithm and calculations described in section 7.3.4.

An hard-coded destination was set in the *PathFinder* module, and with the added Finder block, the simulated robot succeeded in reaching its destination. The output from the updated simulation system proved that the system design, modules and calculations created for the project could actually be used by the simulated autonomous robot to move from one position to another.

9.2.6 Realization of MVP 5: Graphical Simulation

The last implemented minimum viable product iteration was created to satisfy the need of a better way to view the simulated robots movement, and to error check the movement algorithm. Screen-shots from a live simulation session can be seen in Figures 8.64 to 8.70 where the simulated robot moves from a position to a destination using all the prior minimum viable products implemented functionality. To realize the fifth MVP the Control Panel was created and implemented, as described in section 8.1.8.

The implemented graphical simulation serves its purpose of displaying a live simulation of the robot moving to a destination. The control panel displays all the robots physical properties, the status of the modules in the system and provides a convenient way of designating new destinations for the robot with the help of input fields. The canvas where the robot was drawn however could have been better if the cardinal directions and the coordinate system of the screen was translated to that of the real world. E.g. bottom left of screen should have been coordinates 0,0 and an increasing Y value going upwards and an increasing x value going to the right.

9.3 Usability and real world viability

The reason the simulation system was proposed in the first place was to create a system where one can test control software on a simulated robot, before implementing the software on a real one. With a simulation one can easily test different iterations of the software and the algorithm to see if the robot behaves as expected, with no risk of harming hardware or in the worst case humans. But there is no point in simulating the control software if it has to be re-created to fit a robot,

and the testing has to be started anew. To battle this the system design should be as close to the real robot as possible, and with a high modifiability such that it requires small changes to the software to work on the real robot.

The current system design does, in my opinion, fit this description. The use of Reactive Blocks to make modules, and sub-modules (blocks) easily accessible and reusable in addition to using the OSGi framework to make the modules independent and easily replaceable creates a flexible, scalable and highly modifiable system design. It should be easy to add, remove and change parts of the system without having to do larger changes to other parts of the system.

When it comes to the use of the implemented Prototype it does what it should do, it is a proof of concept for the system design and architecture more than a fully functional control system, the short-comings of the control system has been described in the next section. One can argue that a better, more advanced control algorithm using more sensors would have been a good idea. However spending time on this would have been at the expense of other more important parts of the system.

9.3.1 Control Software short-comings

The first thing to notice about the control software is the low amount of sensors used. The control software can move a robot from A to B, but it has no sensors in-place to handle collision detection and to measure position other than a relative position. The robot starts at a given position and all of its calculations and measurements are based on how far the robot has moved away from that position. A slight error in the placement of the robot or an error percentage in the sensors measurements would send the robot in the wrong direction, and there would be no way for it to know or correct the error. Using distance measurement sensors like echo-location or laser would have been a nice addition to the robots sensors, adding the possibility to "see" the world around it. Another possibility would have been to add guidelines with a set distance between them on the floor, where the robot moved, which could have been used to calibrate the robots position.

The database is just used for communication between the Sensor modules and the Pathfinder module, but the data in it could have been used in machine learning. This could be used to map out regular errors in the measurements of the sensors and have the robot ignore them if the other sensors are unaffected. The information in the database could also be used to measure breaking distance, turning radius etc. and make the control algorithm use this to assist the robots driving capabilities.

Chapter 10

Conclusion

The purpose of the master thesis was to use Model-based engineering to create a system design for simulated transport robots. A highly modularized prototype was going to be created using the OSGi framework and the Reactive blocks tool. In this chapter I present a conclusion to the results presented in part III and the discussion in section 9.

The system design does, in my opinion, satisfy the criteria set in the projects description. The design is flexible, scalable, with high cohesion within its modules and low coupling between them. The inter-modular communication blocks created in Reactive Blocks provides an easy way to add functionality, increasing the systems modifiability. The Minimum Viable Product technique used during the research and development periods helped guide the process in the right direction. It helped me focus on the most important parts of the system at an early stage in development and it provided continuous fallbacks throughout the whole project period when moving from one MVP iteration to the next.

The prototype system was created using the Reactive blocks tool(2) and the OSGi framework (3). Using the two technologies together to develop control software had never, as far as I could find, been done before. The merging of the two tools gave an easy, available way to reuse code by implementing the complexity of the OSGi framework in the Reactive Blocks tool. A framework promoting modularization and a tool assisting the development of highly modularized, reusable pieces of code fit together nicely. To use these technologies in the robotics domain, where a lot of concurrent, independent sub-systems control different parts of the hardware in real-time, was in my opinion a great success.

The prototype system served its purpose of being a prototype. The current implementation of the control software cannot be used to control a real life robot with

high accuracy. The current control algorithm is way to dependent on precise sensor input and a near perfect calibration of starting position, to be able to reliably move to a destination.

The goal of the system was to create modules which could be installed, uninstalled, started, stopped and updated during run-time. Something they should do since all modules are implemented as independent OSGi plug-in projects. I had an idea of implementing functionality in the GUI to control the state of the modules, I ran out of time before fully implementing this and unfortunately had to exclude it from the final system.

Chapter 11

Further work

This chapter presents areas which could and should be explored if the work on the system was to be continued.

By building upon the current system design there are a lot of areas which could be experimented on. One could easily add blocks into the simulation loop, subscribing to the current topics and publishing data under new ones. Potential ideas are:

- One can add more versions of the *PathFinder* module in parallel, with different control algorithms and have several *Control Panel* modules running in parallel, displaying the results of the different PathFinders at the same time.
- One can add copies of the already implemented sensor modules and factor in an error on the sensors output to simulate the error real sensors would have on the control algorithm and its effect on getting to the destination.
- One can create new modules, with implemented functionality to send data over the Internet to a real robot with the use of for example MQTT. This could be used to remotely control the engines of a real robot from the simulation system, or the other way around controlling a simulated robot with the use of sensor output from a real robot. Using this technique would fit rather well with the Hardware in the Loop (HiL) proposed in (27), where some individual hardware components are tested in an environment that replaces some parts of the complete system with software simulated components.

These are just some examples of ways to build upon the current system design that I would want to expand on.

It would have been very interesting to create an control algorithm which used

several more simulated sensors, implementing it in the *PathFinder* module and testing it on the *Control Panel*. Sensors I would have liked to implement are; distance measuring sensors like laser and echo-location to measure the distance between the robot and objects in its environment. It would also have been interesting to implement a type of system for the robot to recalibrate its position, for example lines drawn on the floor with a set distance between them and using this in combination with the database and its sensors to implement machine learning. If it has to calibrate the same amount of deviation every-time the robot drives over a line, the robot can use this to adjust its sensor modules.

Implementing the possibility for a sensors output to control the life-cycle of modules would also have been an interesting aspect to research. If an echo-location sensor finds that the distance to the closest object is 10 meters, then it could turn off the Sensor module controlling the laser sensor (laser sensors typically works on close range distance detection).

After creation of a functional more advanced control algorithm I would have tried actually porting the software to a real robot, and testing if the system actually is as flexible as theorized.

Bibliography

[1] Magnus Oplenskedal, *Model-based Engineering of Control Software for simulated robots*

[2] Bitreactive
`http://www.bitreactive.com/` - 2016

[3] OSGi
`https://www.osgi.org/` - 2016

[4] PiBorg
`https://www.piborg.org/diddyborg` - 2016

[5] Raspberry PI
`https://www.raspberrypi.org/` - 2016

[6] Raspbian
`https://www.raspbian.org/` - 2016

[7] Reid Simmons, *"Concurrent Planning and Execution for Autonomous Robots"*

[8] Steven A. Shafer, Anthony Stentz, Charles E. Thorpe, *"An Architecture for Sensor Fusion in a Mobile Robot"*

[9] Erman, L.D., Hayes-Roth F., Lesser, V.R., Reddy, D.R. *"The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty" ACM Computing Surveys 12(2):213-253, June, 1980.*

[10] F. A. Kraemer and P. Herrmann, *"Automated Encapsulation of UML Activities for Incremental Development and Verification," in Model Driven Engineering Languages and Systems (MoDELS), ser. LNCS 5795. Springer-Verlag, 2009, pp. 571–585.*

[11] *F. A. Kraemer and P. Herrmann, "Reactive Semantics for Distributed UML Activities," in Joint WG6.1 International Conference (FMOODS) and WG6.1 International Conference (FORTE), ser. LNCS 6117. Springer-Verlag, 2010, pp. 17–31.*

[12] *F. A. Kraemer, P. Herrmann, and R. Bræk, "Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services," in 8th International Symposium on Distributed Objects and Applications (DOA06), ser. LNCS 4276. Springer-Verlag, 2006, pp. 1614–1632.*

[13] F. A. Kraemer and P. Herrmann, *"Reactive Semantics for Distributed UML Activities," in Joint WG6.1 International Conference (FMOODS) and WG6.1 International Conference (FORTE), ser. LNCS 6117. Springer-Verlag, 2010, pp. 17–31.*

[14] *A. C. Santos et al. Specifying Adaptations through a DSL with an Application to Mobile Robot Navigation. In Proc. of the Intl. Symp. on Languages, Applications and Technologies (SLATE'13), 2013*

[15] `Ritesh Lal and Robert Fitch, A Hardware-in-the-Loop Simulator for Distributed Robotics - 20009`

[16] `Pierre-Emile Duhamel ,Judson Porter, Benjamin Finio, Geoffrey Barrows, David Brooks, Gu-Yeon Wei, and Robert Wood, Hardware in the Loop for Optical Flow Sensin in a Robotic Bee - 2016`

[17] Publish-Subscribe Pattern
`https://msdn.microsoft.com/en-us/library/ff649664.aspx - 2016`

[18] www.sqlite.org
`https://www.sqlite.org/ -2016`

[19] Microsoft Application Architecture Guide, 2nd Edition, Chapter 16: Quality attributes
`https://msdn.microsoft.com/en-us/library/ee658094.aspx - 2016`

[20] Minimum Viable Product Technique
`https://msdn.microsoft.com/en-us/library/ee658094.aspx - 2016`

[21] Image of the DiddyBorg
`https://www.piborg.org/images/DiddyBorg /PiBorg%20DiddyBorg%20Clear%20Raspberry%20Pi%20Robot%20front%201440.JPG`

[22] XLoBorg
`https://www.piborg.org/xloborg/buy` - 2016

[23] *https://osgi.org/javadoc/r4v41/org/osgi/service/event/Event.html - 2016*

[24] *https://osgi.org/javadoc/r4v42/org/osgi/service/event/EventHandler.html - 2016*

[25] Fenglin Han, Jan Olaf Blech, Peter Herrmann and Heinz Schmidt, *"Model-based Engineering and Analysis of Space-aware Systems Communicating via IEEE 802.11"*

[26] Fenglin Han, Jan Olaf Blech, Peter Herrmann and Heinz Schmidt, *"Towards Verifying Safety Properties of Real-Time Probabilistic Systems"*

[27] *R. Isermann, J. Schaffnit, and S. Sinsel, "Hardware-in-the-loop simulation for the design and testing of engine-control systems," Control Engineering Practice, vol. 7, no. 5, pp. 643–653, 1999.*