Collision Avoidance of Autonomous Safety-Critical Real-Time Systems

SCCharts Case Study of a Flying Drone

Felix Machaczek

Bachelor-Thesis
2015

Prof. Dr. von Hanxleden
Real-Time and Embedded Systems
Department of Computer Science
Kiel University

Advised by
Dipl.-Inf. Steven Smyth and Dipl.-Inf. Christian Motika

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

The employment of autonomous unmanned aerial vehicles requires the ability of detecting potential obstacles and avoiding collisions to ensure that neither humans are harmed nor object are destroyed. Therefore it requires a set of qualified hardware and an appropriate software solution. Because the recently published visual synchronous language SCCharts was especially developed for creating such safety-critical real-time applications, it seems to be well-suited for this purpose.

This thesis explains the differences between multiple sensor types, examines the sensor value evaluation, and compares the SCCharts model with a pure C/C++ implementation. To validate the feasibility of a collision avoidance procedure with the help of SCCharts, it describes how to equip an indoor quadcopter with an ultrasonic obstacle detection.

Key words collision avoidance, ultrasonic, safety-critical, real-time system, embedded system, SCCharts, quadcopter, quadrotor, autonomous, flying, UAV, drone, KIELER

For Jana,
my banister in highest height,
my guiding light in darkest night.

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 2
- 1.2 Sequentially Constructive Charts . . . 3
- 1.3 Quadcopter . . . 4
- 1.4 Outline . . . 5

2 Related Work **9**

3 Used Technologies **11**
- 3.1 Arduino . . . 11
- 3.2 3D Printing . . . 11
- 3.3 Remote and Sensor Communication . . . 12

4 Collision Avoidance Design **15**
- 4.1 Choice of Sensor Hardware . . . 15
  - 4.1.1 External Positioning Sensors . . . 15
  - 4.1.2 Internal Proximity Sensors . . . 17
  - 4.1.3 Sensor Comparison . . . 19
- 4.2 Acoustic Distance Measurement . . . 20
- 4.3 Outlier Detection and Signal Smoothing . . . 23
- 4.4 Collision Avoidance . . . 27
- 4.5 Error Handling and Crash Avoidance . . . 30

5 Implementation **33**
- 5.1 Host Code Framework . . . 33
- 5.2 Ultrasonic Sensor Controller . . . 34
- 5.3 Collision Avoidance in C/C++ . . . 36
- 5.4 Collision Avoidance in SCCharts . . . 39
- 5.5 Telemetry Software . . . 42

6 Evaluation **45**
- 6.1 Sensor Behavior . . . 45
- 6.2 Collision Avoidance Implementation . . . 47

7 Conclusion **51**
7.1 Summary 51
7.2 Future Work 52

A Source Code **55**

B Flight Control **57**

C Wiring **59**

Bibliography **61**

List of Figures

1.1 Cyclic, discretized execution of a reactive embedded system 3
1.2 Quadcopter 6
1.3 Sketch of the quadcopter with all hardware components 6

3.1 Arduino IDE with default sketch 12

4.1 Determination of a robot's position with the aid of beacons 16
4.2 Angle of incoming reflection with infrared proximity sensors 17
4.3 Sonic speed in dry air at sea level 20
4.4 There are multiple types of wrong echoes. 22
4.5 Sensor communication between the Arduino Mega and an ulteasonic sensor 24
4.6 Window size delays value processing 25
4.7 Test data from vibrating sensors 27
4.8 Different obstacle distances lead to different forbidden flight directions, marked with a crossed arrow. 29

5.1 Collapsed SCChart 39
5.2 Referenced sensor SCChart 40
5.3 Referenced sensor SCChart 41
5.4 Shortened movement extension in the SCChart with small preview of the whole superstate on the right 41
5.5 Movement verification in the SCChart divided into two regions 42
5.6 Telemetry Software 43

6.1 Rotating the quadcopter only a few degrees can cause the loss of a sensor's echo. 46
6.2 Measured values from one sensor while increasing the distance to a wall with different speed. 46
6.3 Side view on one of the quadcopter's arms next to a tabletop. The sensor does not recognize the table. 47

C.1 Wiring of all electric components 59

List of Tables

4.1 Comparison of Sharp infrared sensors 18
4.2 Comparison of different proximity sensor types 19
4.3 Arduinos digital pins usable for interrupts 22
4.4 Single outlier in constant value series 26
4.5 Single outlier within falling values 26
4.6 Quickly falling values without an outlier 26

Abbreviations and Symbol Index

KIELER Kiel Integrated Environment for Layout Eclipse RichClient

I/O Input and Output

IDE Integrated Development Environment
A software environment for developing and and compiling software.

LGPL GNU Lesser General Public License

GPL GNU General Public License

ESC Electronic Speed Control

UAV Unmanned Aerial Vehicle

MoC Model of Computation

M2M Model to Model

RES Reactive Embedded System

IR Infrared

Chapter 1

Introduction

According to Moore's law[1] the power of computing chips doubles about every 18 months. While particularly for embedded systems hardware limitations were a big problem for a long time [Lee05], even the smallest systems seem to eventually profit from this evolution today. Storage and calculation power have become affordable in small sizes. The results can be seen in most manufacturers' modern cars, for example. Every model's next generation comes with new features and usually relies more on computing power than the predecessor. An article by Klaus Bengler et al. [BDF+14] gives an interesting overview about the development of driver assistance systems since the 1980s. It retrospects the past years and makes assumptions about probable systems in the future. We can see an obvious tendency towards an excessive use of computers, and this example shows a change of priorities when developing modern technical systems for consumers. Especially with autonomous real-time systems this leads to new challenges.

Thinking of a driverless automotive in road traffic, although autonomous navigation isn't a technical problem anymore nowadays, a self-operating vehicle will always be confronted with unpredictable obstacles[2]. While cars do mostly operate in a familiar environment, there will be an increasing number of autonomous systems in completely unknown settings. Imaginable are for example various kinds of Unmanned Aerial Vehicles (UAVs) for surveillance, parcel delivery or rescue operations in dangerous situations. Such systems could be started anytime at any place and need to find a way avoiding obstacle collisions. With a growing demand for such autonomous systems in the coming years there is a need for fast, precise and, first of all, safe embedded systems.

As humans we are capable of detecting our environment and reacting to it in a proper way nearly instantly. When constructing a standalone machine and giving it the ability to interact with its surroundings we need to think about two main aspects: timing and correctness. Most commonly known computing systems such as notebooks or smartphones are built to accomplish a task with an expected expected outcome, while a jitter in execution time is mostly irrelevant. The latter does not necessarily apply to a real-time system. Depending on its application it may stick to a set recurring task frequency or

[1] Gordon Earle Moore, *1929

[2] Gomes, Lee. (2015, Mai 15). Urban Jungle a Tough Challenge for Google's Autonomous Cars [article]. Retrieved from http://www.technologyreview.com/news/529466/urban-jungle-a-tough-challenge-for-googles-autonomous-cars/

provide results in a tight time limit. While the real world is existing in a constant time flow any computing machine has an inner clock, which confines its number of calculations per time unit. Moreover, an embedded real-time system's calculation power may be limited by an especially compact size or a very small energy consumption.

The common approach on how to work with real time in such a system is a repeatedly called sequential function, which performs the task of the system. In the UAV example such a timing-sensitive and frequently performed task is the stabilization routine. The more often the flight controller can correct a deviation from a desired angle the more steady it can fly. Getting distance values of multiple sensors can be time consuming, depending on the used sensor type. A too long delay in the flight controller could lead to a destabilization and cause a loss of control or even a crash. This brings us to the second mentioned aspect, the correctness of the system, which comprises the guaranty of safety. When operating in the air a problem could cause a fall and lead to potential damage to the system. Particularly when operating in public areas, much more important than avoiding destruction of the UAV is to prevent any person from getting harmed. Controlling a UAV implicates multiple safety-critical problems:

- An unintended movement towards a wall or another obstacle can cause physical contact and provoke the burst of a rotor and possibly a motor. Also with rotor protectors the UAV may wedge, overturn and cause even more damage.
- Using a remote connection involves the danger of a connection loss and potentially faulty instructions due to misinterpretation.
- Hovering is extremely energy consuming. Depending on the battery capacity the flight time of a UAV can be rather short. In case of low voltage the function of all components cannot be ensured.
- A program bug such as an endless loop or even a system hang up would cause the flight controller to stop working. This would lead to a fatal crash.

1.1 Problem Statement

Building an autonomous UAV means to reconcile both, being in time with all essential operations and ensuring the safety of the system and its environment. All points noted above need to be implemented without interrupting the stabilization routine. They include watching over the operation of the system, reading out sensor values as often as possible and designing an algorithm that observes all directions concurrently. This is qualified for a case study on the feasibility of an obstacle collision avoidance system for a flying drone in the synchronous language SCCharts [HDM+14]. For this purpose a quadcopter is assembled, equipped with appropriate hardware for obstacle detection and programmed

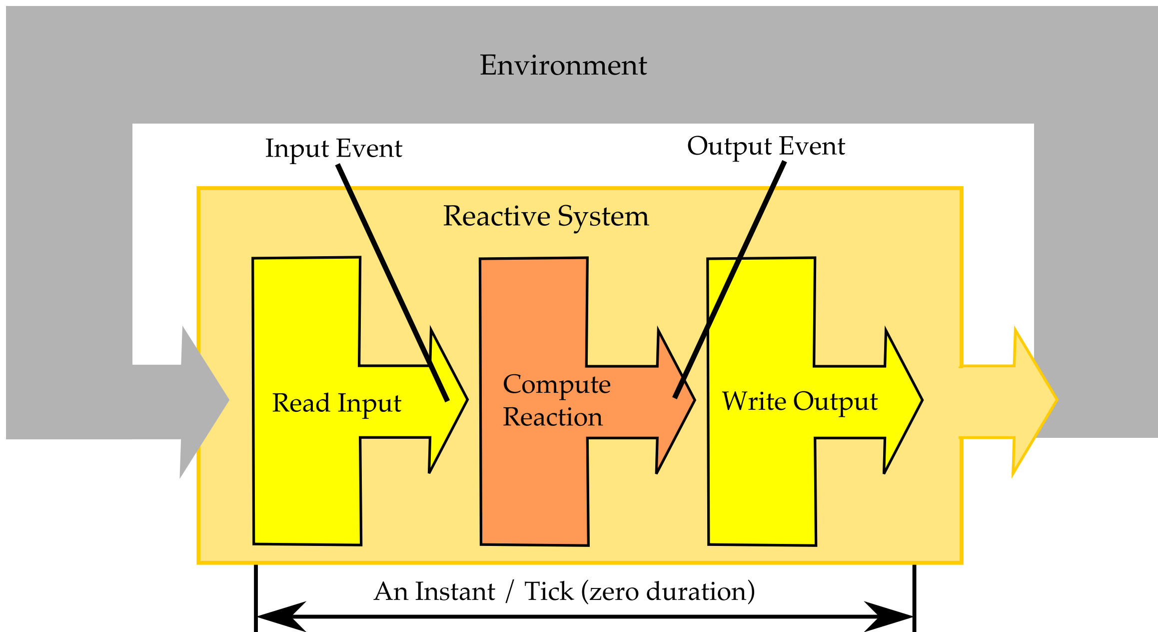


Figure 1.1. Cyclic, discretized execution of a reactive embedded system [MHH13]

with all necessary software to run the SCChart. The problems of different sensor types will be examined as well as data communication and safety when remotely controlling a vehicle. Other required functions like hovering, landing and simple flight, which are developed simultaneously in the context of the project this thesis is part of, will be assumed as existing.

1.2 Sequentially Constructive Charts

SCCharts is a visual synchronous language with statechart notation [HDM+14]. Its key benefit to similar languages like SyncCharts [And96] is its Model of Computation (MoC), which allows deterministic concurrency. While SyncCharts forbid different values of a signal within one macro tick, this is allowed and intended in SCCharts. Figure 1.1 shows the tick execution of an Reactive Embedded System (RES) where several input signals are read from the environment and after computation some outputs are written back to it. Its causality makes SCCharts ideal for modelling RESs because any signal may have different input and output values.

There are two types, core and extended SCCharts. Core SCCharts consist of regions which contain simple and hierarchical states. States can be connected via multiple types of transitions. Primitive data types may be used. Extended SCCharts are built on top of the core version and are often described as *syntactical sugar*. They additionally offer features

like entry, during and exit actions for states, delays, data-flow or conditional termination. Via Model to Model (M2M) transformation any extended SCCharts can be converted to core SCCharts [MSH14].

For developing SCCharts it is recommended to use the Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) SCCharts editor and compiler. A quick start guide can be found on the official website of the KIELER project[3]. The included compiler is able to translate any core or extended SCCharts to C or Java code. To reproduce the macro tick behavior of the underlying statechart the resulting code is divided into two functions — the *reset* function and the *tick* function. Initially the *reset* function needs to be called to initialize all variables whereupon every *tick* call performs one macro tick in the state machine.

Because the SCCharts language is still in development there is a need for studies about the feasibility of different projects' approaches.

1.3 Quadcopter

The quadcopter does not result from an assembly kit but is completely self constructed. There are multiple hardware parts bought from different vendors and other parts printed with a 3D printer. We decided to use the F330 Glass Fiber Mini Quadcopter Frame as a base, because it is not too large for indoor flights and not too small to provide enough space to attach a microcontroller and suitable hardware for collision avoidance. To get sufficient lift during flight four Turnigy Aerodrive SK3 2822-1275 Brushless Outrunner motors are applied. Each motor gets its electric power from and is driven by an own Electronic Speed Control (ESC) which on its part is controlled by the flight controller and supplied by a Turnigy 3S 20C Lipo battery pack with a capacity of 2200mAh. While hovering this brings a flight time of approximately eight to ten minutes.

For the flight controller we chose an 5V 16 MHz Arduino Mega 2560 though there exist multiple other microcontroller manufacturers such as BeagleBone or ST. At the project's starting point the Arduino was already available and due to their popularity Arduino boards have a large community. Besides the main microcontroller one Arduino Nano and one Arduino Pro Mini are employed to process the collision avoidance sensors' signals (see also Section 4.2). By using an MPU-9150 9-axis motion tracking chip the flight controller is able to get acceleration data to hold the quadcopter stable. As explained later in Section 4.1 for the distance measurement several HC-SR04 ultrasonic sensors are being used. They observe the environment for potential obstacles.

To provide a wireless communication interface we built in an Aukru HC-06 Bluetooth RF Transceiver module. This enables the flight controller to receive commands and send important data. With the specially developed telemetry software for desktop computers

[3] http://rtsys.informatik.uni-kiel.de/kieler

a user is able to control the throttle of the motors and the flight direction to simulate an autonomous movement. In case of an oncoming collision the flight controller will overwrite the user's commands and decide to avoid it. However, in case of a problem an emergency landing can be instructed. All required information for debugging reasons can be displayed in a log view. For further information about flight control see Appendix B. The wiring diagram of all components can be found in Appendix C.

As it can be seen in Figure 1.2 during this project the quadcopter was repeatedly expanded with 3D printed parts made out of red filament.

- To tightly mount the Arduino Mega 2560 a holder was printed, which could be screwed to the frame.
- For safe starting and landing reasons it was decided to print a pair of feet which we found on MakerBot Thingiverse[4].
- To guard against potential damage during test flights, rotor protectors extend the arms of the quadcopter.
- Additionally every ultrasonic sensor has a printed holder via which it can be applied to the copter.

A visual overview about all attached hardware gives Figure 1.3.

For further understanding the front of the quadcopter is between both red arms and the ultrasonic sensors are numerated from 0 to 9 as *Front*, *Front-Right*, *Right*, *Back-Right*, *Back*, *Back-Left*, *Left*, *Front-Left*, *Top*, and *Bottom*.

1.4 Outline

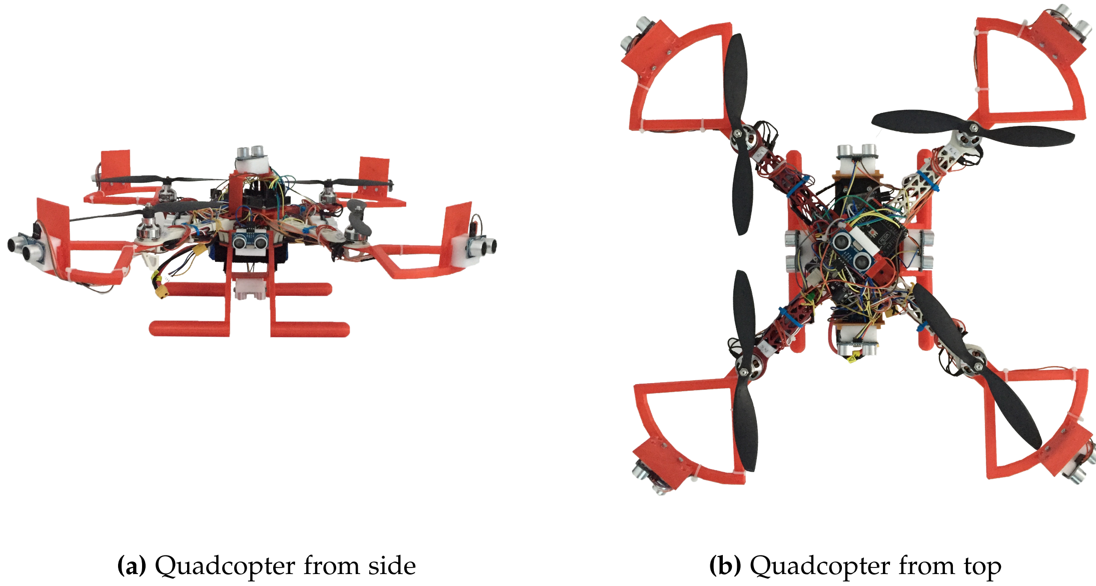
This Section gives an overview about the further course of this work.

In Chapter 2 related work to this thesis is discussed regarding the choice of obstacle detection hardware and collision avoidance algorithms.

Chapter 3 presents all notable technologies that were required to archive the project's goal, such as the Arduino platform and third party libraries for sensor communication.

All conceptual topics of this thesis are covered in Chapter 4. It contains the evaluation of distance measurement hardware and decision for one type, the required theoretical background of distance measurement and furthermore the preparation of raw sensor data. Subsequently, it describes the concept of collision avoidance and the handling of potential error sources.

[4]DJI F330 Quadcopter landing gears. By diegolopmon. Published 2015, March 23. Retrieved from http://www.thingiverse.com/thing:736947



(a) Quadcopter from side **(b)** Quadcopter from top

Figure 1.2. Quadcopter

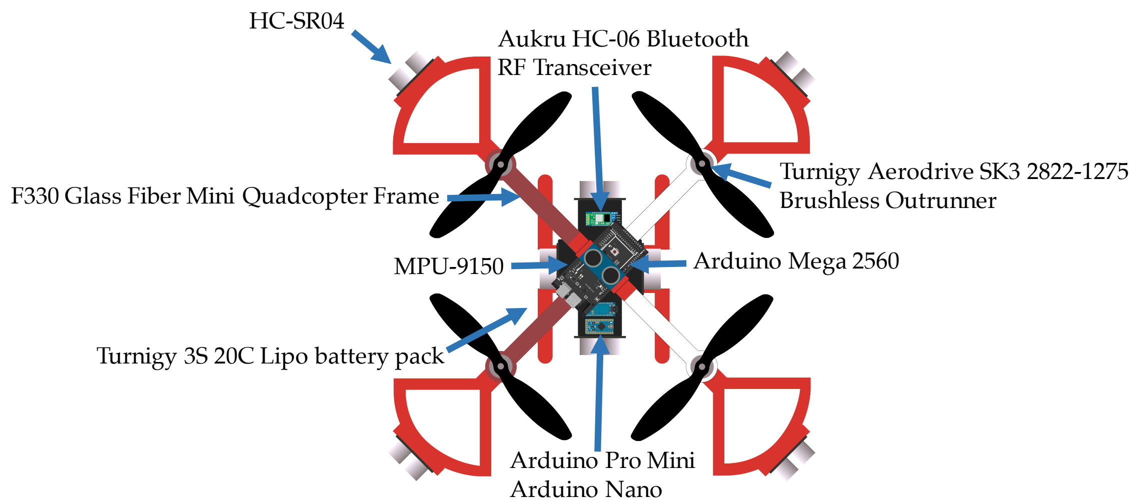


Figure 1.3. Sketch of the quadcopter with all hardware components[a]

[a]This image was partly created with the help of Fritzing (http://fritzing.org/). The same applies to the following ones, that show the Arduino Mega 2560, the Arduino Nano or the Arduino Pro Mini.

In Chapter 5 the important parts of the written software are outlined. It is shown how the SCCharts' generated code is run on the microcontroller and how the distance sensors are driven and processed. The collision avoidance implementation is written in C/C++ and in SCCharts for debug and comparison reasons. Pieces of both versions will be presented to understand the benefits and drawbacks later on. In addition the developed telemetry software is introduced.

Following concept and implementation, Chapter 6 compares both emerged versions of software and the feasibility of the intention.

To come to a conclusion Chapter 7 summarizes this thesis and gives a perspective on future work.

Chapter 2

Related Work

In the field of collision avoidance there exists some work, that is relevant for this thesis. The following describes different approaches and their relation to this paper.

Bouabdallah et al. [BBP+07] describe the development of a collision avoidance system for the OS4 quadrotor helicopter, created in the Autonomous Systems Lab (ASL) Zürich, Switzerland from 2003 to 2005. OS4 is equipped with an x86 compatible mini computer with a 266 MHz Geode 1200 processor and a Debian-based lightweight Linux distribution and at the beginning of the work the quadcopter is already able to hover stably over ground. While hardware limitations have become less nowadays, this computer has calculation power many times that of the capabilities of our Arduino boards. The preconditions of the mentioned project are completely different and cannot be compared. In the article the procedure of applying unnamed ultrasonic sensors is outlined and every collision avoidance procedure approach is simulated with MatLab / Simulink. Therefore, two assumptions are made for flight maneuvers:

- ▻ The quadcopter holds its altitude during the whole tests.
- ▻ It can only fly in the four directions where the ultrasonic sensors are applied.

The paper lists five approaches for obstacle avoidance, which partly build on each other. The first and basic approach is to define two threshold values which divide three sensor states "Far", "Close" and "Too close" which determine the further flight movements. Following approaches try to make the avoiding movements more natural and fluid. The three state classification is a good approach as it is used in other works, too. It will be the inspiration for the collision avoidance procedure in this thesis.

Gageik et al. [GMM12] also start with an existing quadcopter from the Aerospace Information Technology department of the University of Würzburg by attaching 12 redundant SRF02 ultrasonic sensors for a 360° circle. Similar to Bouabdallah et al. they make a division for each sensor into three zones, green for safe zone, yellow for close zone, and red for dangerous zone and state that this behavior is best described by a state machine, which comes close to the SCCharts idea implemented in Chapter 5. Gageik et al. use self-processing ultrasonic sensors which do not require extra distance calculation on controller side, but give a fair warning against possible interferences of the individual sensors.

Benz [Ben13] builds on top of the work by Gageik et al. and describes the need for a different measurement method and a higher value production rate than the ultrasonic sensors can provide. He mounts supplementary infrared sensors and uses a combination of filters to compare both sensor system results. As an evaluation Benz shows an elaborate setup for testing the collision avoidance procedure on a rollable undercarriage in an artificial environment with even a dummy for mimicking a human. If enough funding is available the use of multiple sensor types is a good idea. This cannot be done in this project but both technologies are evaluated against each other.

Altuğ et al. [AOT03] propose the approach of a two camera vision based quadcopter control. One camera is attached to the quadcopter itself and one camera stands on the ground as an observer. Combining both visual inputs they show that this method is possibly more effective than other position detection systems, especially indoors where no GPS signal is available and proximity sensors might return blurry results. This method of position and resulting obstacle detection is evaluated in Section 4.1 against other options.

Rahman et al. [RAE14] use the here applied non-processing HC-SR04 ultrasonic sensors, not for an aerial vehicle, but for a vehicle on ground. They have a separate microcontroller next to their main navigating microcontroller for handling the sensors. It presents a similar solution for ground vehicles.

While other related work mostly relies on an existing UAV, within the project around this thesis the quadcopter has to be self-assembled and a flight control needs to be developed. Alhough there exist papers with non-processing sensors like the in this work chosen HC-SR04, no approaches on implementing both a stabilization routine and a sensor evaluation for a UAV simultaneously could be found. More calculation power or self-processing sensors make it easier to concentrate on the actual avoiding algorithm while this thesis is in large part focussed on solving the real-time problem with non-processing, slow sensors beside a quick and frequently safety-critical function.

Chapter 3

Used Technologies

3.1 Arduino

Arduino, also known as Genuino since May 2015, is an open source hardware and software development platform under which a bunch of different microcontroller boards and compatible periphery were produced. While there exist many Arduino clones from different manufacturers, today all originals and copies share their key features - an easy to program controller and several digital and analog Input and Output (I/O) pins. To program an Arduino compatible controller the free to download Arduino Integrated Development Environment (IDE) can be used. It provides the necessary libraries and headers, which are automatically added to any project in the background, compiles and uploads an Arduino program to a selected board with a click.

The Arduino language is based on C/C++ and provides some additional functions like working with the I/O pins, writing and reading data to and from a serial connection and other hardware specific actions but also misses some usually known libraries like the C++ std library due to a limited memory capacity.

An Arduino project is called sketch and represented by a folder and a *.ino* file, both with the sketch's name. Any sketch must contain this main file, but may contain multiple other *.ino*, *.c* or *.cpp* files.

Figure 3.1 displays a screenshot of the Arduino IDE with a default sketch opened. The main *.ino* file contains analogous to an SCCharts' generated code a *setup* function and a *loop* function. *setup* is called initially when the board is powered on and *loop* gets triggered periodically afterwards in an endless loop.

3.2 3D Printing

Tinkercad[2] is a popular online 3D modeling tool founded in 2011 and later acquired by Autodesk in 2013. After a registration any user can create, store and modify 3D models. Unlike other professional modeling tools by Autodesk the usage of Tinkercad does not require any special knowledge and can be managed even by amateurs. Besides an online sharing platform Tinkercad provides the ability to download created models for 3D

[2]https://www.tinkercad.com/

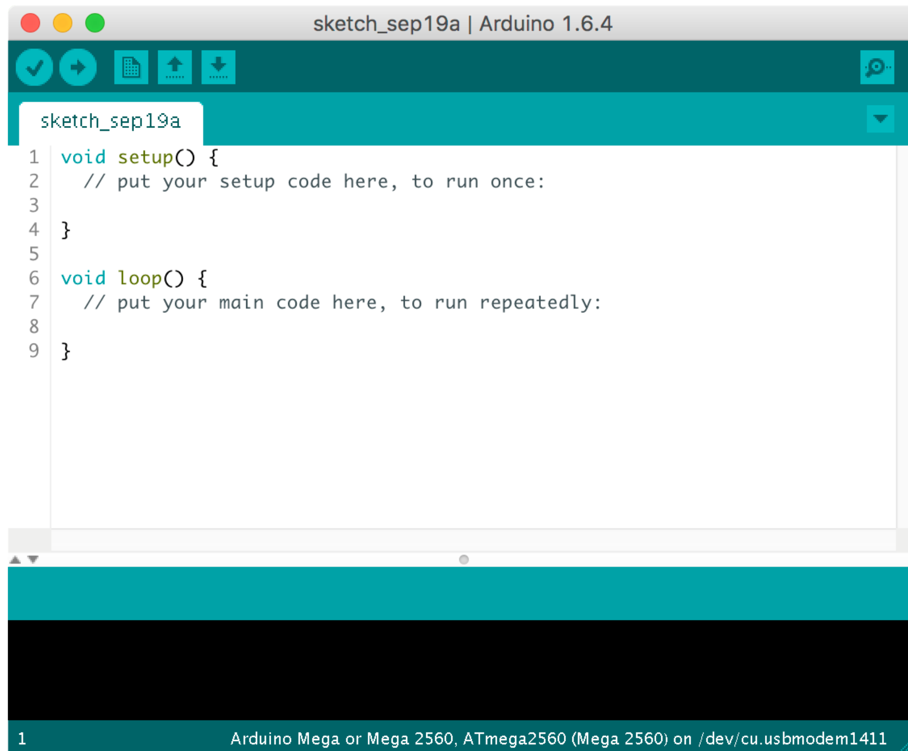


Figure 3.1. Arduino IDE[1]with default sketch

printing as .STL, .OBJ, .X3D or .VRML files. For these reasons Tinkercad was used within this project to create printable models to extend the quadcopter.

For printing the created models a 3D printer extruder was involved. With this technique thin layers of melted plastic filament are deposited above each other. This gives the ability to create compact three-dimensional objects. By using such a printer it was possible to print legs, a holder for the Arduino and - most important for this work - the holders for the later on introduced proximity sensors.

3.3 Remote and Sensor Communication

Serial port is a communication method between computers and periphery where each bit is transferred one after another. While for a serial connection usually a wire is required, the Bluetooth Serial Port profile can emulate a serial cable by radio.

[2]https://www.arduino.cc/

Using Bluetooth Serial Port allows the remote connection from a computer or mobile device to the HC-06 Bluetooth module and communicating with the Arduino as if it was connected via cable.

As stated in the Serial Port Profile Specification[3], the RFCOMM protocol is used on transport layer between two devices, which in turn builds on top of the reliable L2CAP protocol. Due to this characteristic a verification of sent values is redundant.

jSerialComm, originally serial-comm, is a modern Java library for serial port communication by Will Hedgecock under GNU Lesser General Public License (LGPL). Its benefits are its simple way of use and the platform-independency, meaning it will automatically load the correct native libraries depending on the running environment.

Beside the wireless communication to a ground station the Microcontroller needs to get data from the proximity sensors. NewPing[4] for Arduino is a library by Tim Eckel under GNU General Public License (GPL) license, which extends the default abilities when working with most common ultrasonic sensors including the HC-SR04, which is used within this project. Unlike most ultrasonic measurement implementations this library does not use the Arduino's *pulseIn* function, but a hardware timer interrupt.

[3] http://bluetooth.org, BLUETOOTH SPECIFICATION, Serial Port Profile, Revision: V12, Date: 2012-07-24

[4] https://bitbucket.org/teckel12/arduino-new-ping/wiki/Home

Chapter 4

Collision Avoidance Design

Unlike a vehicle on the ground a UAV does not have the chance of touching an obstacle and afterwards trying to get away from it, but needs to avoid any physical contact whatsoever. Avoiding collisions with obstacles requires three logical steps:

1. Detecting a potential obstacle with suitable hardware before a collision can happen.
2. Prohibiting movements towards the obstacle in order not to navigate into a more dangerous situation.
3. Moving away from the obstacle if no other collision is provoked.

In the following all three points are covered. This includes the evaluation of different sensor types, the correct handling of the sensor outputs and a mathematical approach of collision avoidance.

4.1 Choice of Sensor Hardware

To detect possible obstacles around a UAV a set of sensors observing the environment is required. We can differentiate between two kinds of mechanisms, external position detection and internal proximity detection.

4.1.1 External Positioning Sensors

External positioning sensors are sensors, that are not attached to the quadcopter directly but in it's environment. While external detection can provide a rather precise result in a known environment, it involves a predefined and accurate hardware setup. Therefore, autonomous flight is only achievable in a limited way with respects to the accessible area.

Beacon Position Detection

Beacon position detection is a way of determining the relative position of a system to a number of beacons and calculating the absolute position in a room with possible known. The system sends a request signal to all listing beacons, which send their signals back. By time measurement the system is able to calculate the distances between itself and

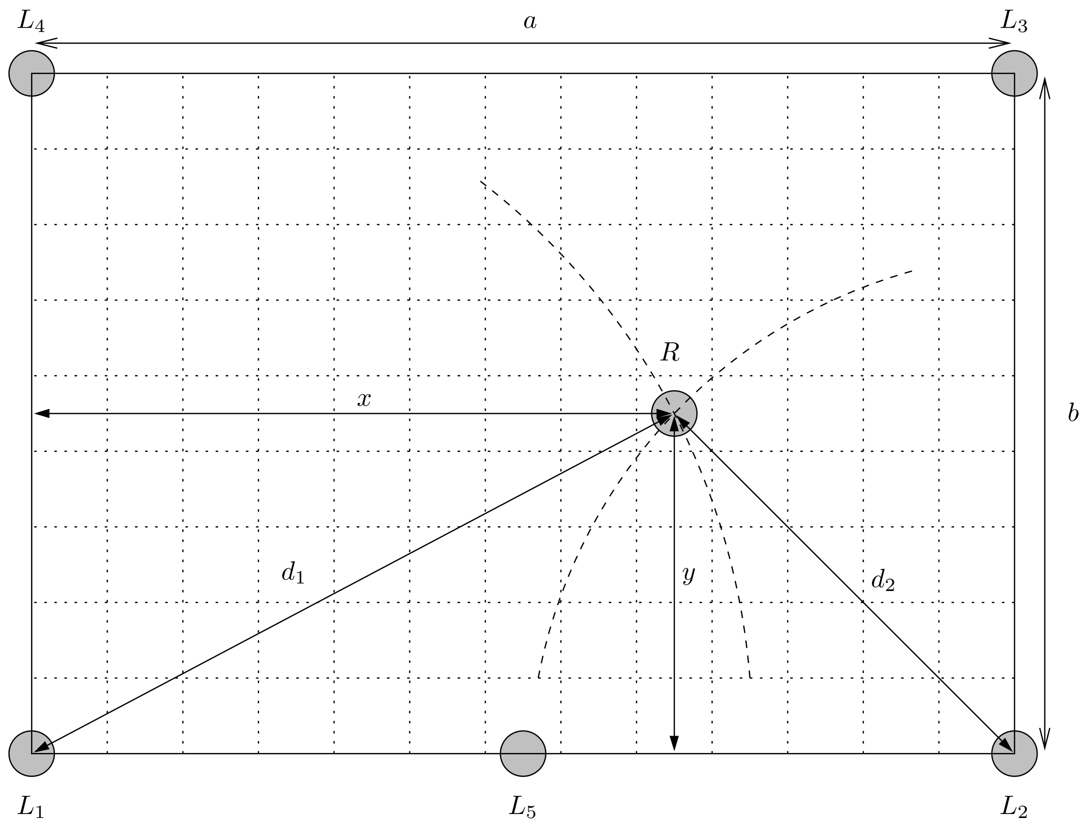


Figure 4.1. Determination of a robot's position with the aid of beacons [Höh06]

all senders. As shown in Höhrmann's research project [Höh06], Figure 4.1 outlines the determination of a robot's (R) position between five beacons (L_1 to L_5). When knowing the propagation speed of the signal type, the distance to one of the beacons can be calculated by measuring the time it takes to send a signal to one of the beacons and getting back the answer. If the distances to at least two beacons is present, it is possible to calculate x and y with help of the Pythagorean theorem.

This method of position detection needs a precise buildup of beacons and a good knowledge of the room between and around those. If all necessary information is present, the system might calculate its exact position in the room and avoid collisions with potential obstacles. Of course, this is only possible in a fixed environment with known positions of all objects and our system could not react to sudden changes without reprogramming.

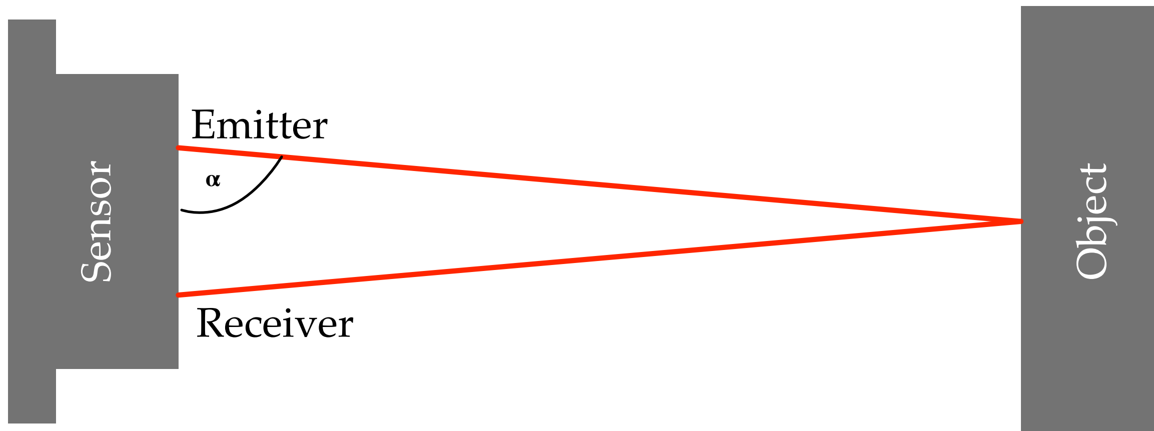


Figure 4.2. Angle of incoming reflection with infrared proximity sensors

Position Detection via Image Processing

To deal with a changing environment another approach is the use of multiple cameras, as it is presented by Altuğ et al. [AOT03] for example. With the help of an image processing software a 3-dimensional model of our system and the room can be created to navigate the system safely, even with unpredicted obstacles. The use of such a video positioning system would be rather sensitive to the lighting conditions and require external calculation capacity.

4.1.2 Internal Proximity Sensors

Internal proximity sensors are directly applied to the quadcopter. Their returned data does not describe the copters absolute position in a room, but instead the relative distance to potential obstacles or walls. The amount and alignment of the sensors have a determining influence on the quality of the obstacle detection.

Infrared Proximity Measurement

An Infrared (IR) sensor can detect an object in a very short period of time because it relies on the speed of light. For that purpose an emitter diode emits an IR light pulse, which is reflected by the regarding object and afterwards caught by a special lens to ascertain the incoming pulse's angle. As shown in Figure 4.2, the emitter, the object and the receiver form a triangle, with which the distance to the object can be calculated via triangulation. The drawback of this technique is that most sensors are built on a special lens and a CCD array to detect the incoming light's angle. This makes IR sensors expensive. As a result most common sensors have a set range in which they can detect an object. The popular

Sharp IR sensors[1] for example exist in four versions from all together 4cm to nearly 5.5m, while each sensor only covers a specific range.

Table 4.1. Comparison of Sharp IR sensors[2]

Model	Min. distance	Max. distance
GP2D120/GP2Y0A41	4 cm	30 cm
GP2Y0A21	10 cm	76 cm
GP2Y0A02	20 cm	150 cm
GP2Y0A710	91 cm	549 cm

Table 4.1 lists all available analog IR sensors by Sharp and shows that no sensor is capable of observing a complete area next to the quadcopter. Using IR for distance measurement over larges ranges would require multiple sensors of different kinds per measure point, which would lead to interferences and measuring errors. Furthermore sun light covers a wide spectrum of wavelengths including IR light and can interfere with the light pulse of an IR sensor. This makes the sensor hardly usable outside. Black or very dark objects may absorb the light pulse, which makes the sensor also sensitive to colors.

Laser Proximity Measurement

Another optical distance measurement method is the laser distance measurement. For the proximity determination the time between sending a laser pulse and receiving its reflection is measured. With the constant speed of light the distance between the sensor and the object can be calculated. This requires a very fast chip, especially for short distances of a few meters. There exist only a small amount of sensors which are small enough to be practical with a quadcopter of ours size, but for a much higher price than all other sensors mentioned in this section. As an example the LIDAR-Lite v2[3] can be named.

Furthermore, we should not ignore the danger of potential eye damage when using an autonomous system equipped with laser emitters in all directions.

Ultrasonic Proximity Measurement

Ultrasonic proximity measurement does also rely on time measurement between sending a pulse and receiving its echo. Ultrasonic sensors come in two performances, self-processing sensors and non-processing sensors. Self-processing sensors have a chip built in which starts measurements repeatedly, calculates the distance between sensor and object on its

[1] Acroname (2015, September 19). Sharp Infrared Ranger Comparison. Retrieved from https://acroname.com/articles/sharp-infrared-ranger-comparison
[2] See footnote 1.
[3] https://cdn.sparkfun.com/datasheets/Sensors/Proximity/lidarlite2DS.pdf (2015, Sep 24.)

own, and returns the determined proximity. Non-processing sensors only provide the required digital pins to create an acoustic pulse and to detect the pulse's echo. Because they are not able to calculate any data the calculation needs to be done eternally. Because sonic speed is about $340\,\mathrm{m\,s^{-1}}$, travel times can be measured by a simple microcontroller. While ultrasonic sensor measurement is relatively slow compared to distance measurement with light, the big advantage of non-processing ultrasonic sensors is their simplicity which make them very cheap in price.

Analogous to optical proximity detection acoustic distance measurement does not work well with sonic absorbing objects such as clothes or a curtain.

4.1.3 Sensor Comparison

Table 4.2 compares the considered proximity sensor types on the basis of range, speed, field of vision and price.

Table 4.2. Comparison of different proximity sensor types

Sensor type	Range	Speed	Field of vision	Price
Infrared	small	fast	wide	moderate
Laser	large	fast	narrow	expensive
Ultrasonic (processing)	medium	slow	medium	cheap
Ultrasonic (non-processing)	medium	slow	medium	moderate

Because a UAV is not able to stop the most important criterion for the sensor hardware is the range in which it can detect objects. Farther sight means a higher level of maneuverability and a faster flight. Although IR sensors are fast in getting results and have a wide visual field their small range makes them unusable for monitoring the surroundings of a UAV alone. Whereas laser measurement can provide a very large range their narrow field of vision and especially their high price make them also not constructive for the intention. The narrower the visual field of a sensor is the higher is the amount of required sensors to cover enough space around.

With only ultrasonic sensors remaining it's a question of expenses which sensor type to choose. While one self-processing *PING))) Ultrasonic Distance Sensor*[4] is at about 30$ at the time of writing, for this price 15-20 non-processing *HC-SR04* sensors can be ordered. As this work is a case study and should provide the possibility for further extensions at this point the decision is made for the non-processing sensors.

[4]https://www.parallax.com/product/28015

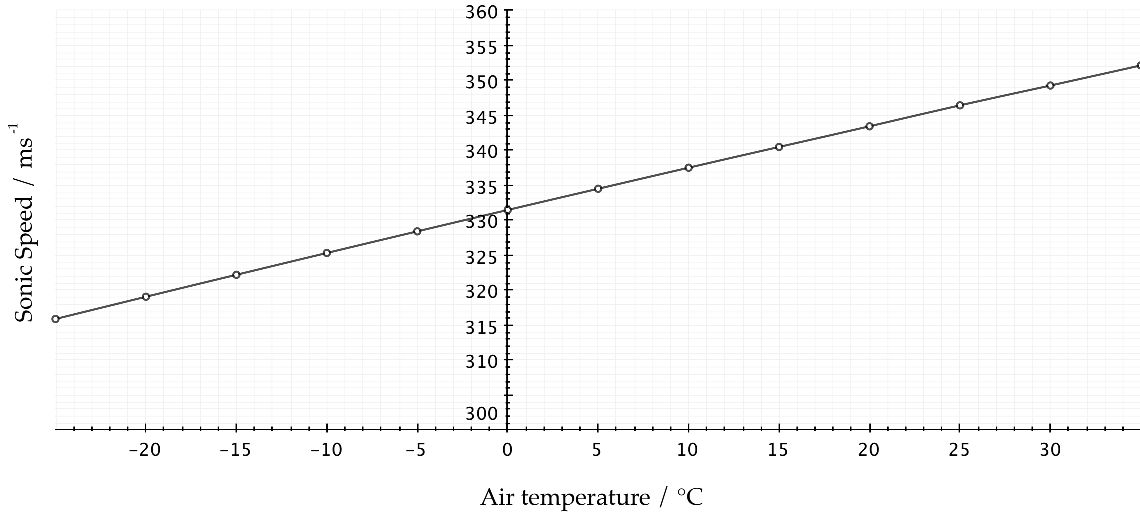


Figure 4.3. Sonic speed in dry air at sea level

4.2 Acoustic Distance Measurement

The HC-SR04 ultrasonic sensor has four pins of which two are for power supply and two for the measuring. It waits for a short *HIGH* signal on the *Trig*-pin, then generates an ultrasonic pulse and starts listening for its echo to return. While listening the sensor sets the *Echo*-pin on *HIGH*, so a negative edge to *LOW* on this pin means that the echo has returned. Detecting an obstacle's distance means measuring the time from the pulse's generation to its echo's return:

$$d = \frac{t \cdot c}{2} \tag{4.2.1}$$

where c is the sonic speed, t is the measured time and d is the distance to the obstacle.

Assuming an indoor flight with dry air at about sea level and 20°C, the sonic speed is about $343.2\text{m}\,\text{s}^{-1}$, as sketched in Figure 4.3. We can translate this to $\mu\text{s}\,\text{cm}^{-1}$:

$$343.2\text{m}\,\text{s}^{-1} = \frac{1}{343.2}\text{s}\,\text{m}^{-1} = 2913\mu\text{s}\,\text{m}^{-1} \approx 29.1\mu\text{s}\,\text{cm}^{-1} \tag{4.2.2}$$

This gives us the usually used form:

$$d\ [\text{cm}] = \frac{t\ [\mu\text{s}]}{2} \cdot \frac{1\ [\text{cm}]}{29.1\ [\mu\text{s}]} \tag{4.2.3}$$

To implement such behavior the usual approach by the Arduino community is to just trigger one sensor after another and wait for the negative edge each. This can be realized like in Listing 4.1.

```
1 // for each sensor {
2   long duration, distance;
3   digitalWrite(trigPin, HIGH);
4   delayMicroseconds(10);
5   digitalWrite(trigPin, LOW);
6   duration = pulseIn(echoPin, HIGH);
7   distance = (duration/2) / 29.1;
8 // }
```

Listing 4.1. Often recommended approach for working with the HC-SR04 sensor on Arduino

The *pulseIn* function is a blocking function which measures the time a signal has a specified value. This leads to timing problems when using this snippet in the flight control. Lets assume, the quadcopter is somewhere in the middle of a large room and neglect the distances to the ground and ceiling. The Arduino has to measure the proximities of eight sensors. In the equation (4.2.2) we can see that the sonic needs 2.9ms per meter. A distance of two meters in all directions can add up to $8 \cdot 2 \cdot 2\text{m} \cdot 2.9\text{ms}\,\text{m}^{-1} = 92.8\text{ms}$. With a distance of three meters for all sensors the processing time would be approximately $8 \cdot 2 \cdot 3\text{m} \cdot 2.9\text{ms}\,\text{m}^{-1} = 139.2\text{ms}$. Having the stabilization routine operating with tick times under 10ms, this measurement method cannot be done during one tick.

Another problem with the *pulseIn* function are lost echoes. *pulseIn* performs busy waiting and is predefined to delay further processing up to three minutes[5]. In case a sensor points towards an unfavorable direction the echo could not return to the sensor which would lead to an even longer waiting time. Figure 4.4 sketches multiple echo types including lost echoes.

As the same time it also shows ghost echoes. Ghost echoes are echoes that are reflected over multiple objects or walls either to the emitting sensor, which then measures a larger distance than it is actually present, or even another sensor, that recognizes and object where no object is existing. While lost echoes only bring problems with sonic swallowing objects, there is no possibility of handling ghost echoes, which pretend to come from a close object that is not there. The only way of averting this problem is to keep the delays between triggering the particular sensors large enough to preempt the subsist of the sonic pulse from another sensor when starting a measurement.

The first approach to circumvent this problem is to trigger one sensor and afterwards check in every tick if the *Echo*-pin is still *HIGH*, and if not take the time. In the equation (4.2.2) we can see that sonic needs about 29.1µs per cm, which gives us per millisecond tick time an inaccuracy of $1000\mu\text{s}/29.1\mu\text{s}\,\text{cm}^{-1} = 34.4\text{cm}$. Thus a distance measurement following this approach is unrewarding.

[5] https://www.arduino.cc/en/Reference/PulseIn
[6] https://www.arduino.cc/en/Reference/AttachInterrupt

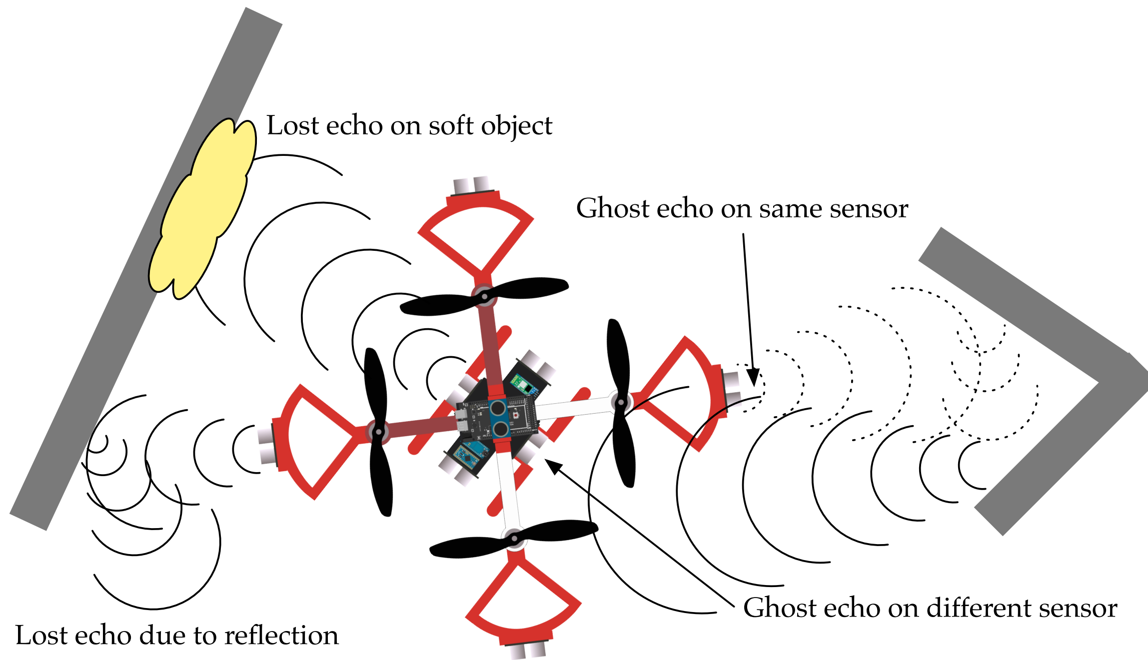


Figure 4.4. There are multiple types of wrong echoes.

Table 4.3. Arduinos digital pins usable for interrupts[6]

Board	int.0	int.1	int.2	int.3	int.4	int.5
Uno, Ethernet	2	3				
Mega2560	2	3	21	20	19	18
32u4 based (e.g Leonardo, Micro)	3	2	0	1	7	
Due, Zero	More pins					

Like most microcontrollers an Arduino is able to receive hardware interrupts on specified pins. Table 4.3 compares different Arduinos for this purpose. The number of interrupts for the Arduino Mega 2560 is limited to six, while two of these are reserved for the I2C protocol, which is used for the gyroscope communication.

To handle these issues Tim Eckel wrote the NewPing library for Arduino[7] as introduced in Section 3.3. This library makes use of a hardware interrupt timer to check every 24µs if the sensor's echo has returned. It is possible to define a maximum distance, after which the measurement is canceled and 0 for "out of range" is returned.

[7]https://bitbucket.org/teckel12/arduino-new-ping/wiki/Home

Although the NewPing library is a solution for the timing problems mentioned above, a permanent interrupt of the tick function would still stretch its time significantly. To have the tick times steady and short to guarantee balancing a good solution is to process the sensors externally on a different microcontroller and send new values to the flight controller. That way even for future changes the proximity measurement is independent from the balancing and the safety-critical operations.

To handle all ten sensors one Arduino Nano and one Arduino Pro Mini are used while each of them is responsible for five sensors. This results from the pin limitations of the Arduinos. Whereas the Arduino Pro Mini is smaller, the Arduino Nano provides a 5V output to supply the Bluetooth module. The whole wiring diagram can be found in Appendix C. Figure 4.5 illustrates the communication between the main Arduino and one of the ultrasonic sensors via the second Arduino. The smaller Arduino Nano for the most part does exactly the same. With the help of the NewPing library it triggers one sensor, waits for the echo to return, and calculates the distance to the object. If a predefined time is exceeded without an echo, the Arduino proceeds with the next sensor. After all sensors have been processed it starts again with the first sensor. In case the larger Arduino Mega is ready to receive new sensor data it sends the Arduino Nano an interrupt. Before the latter continues its measurement it sends the latest sensor values to the Arduino Mega and confirms afterwards with an interrupt on its side. The interrupts help to communicate between both systems without waiting for each other. All data sent over the serial port, is buffered on hardware side until it is read.

The HC-SR04 sensors have a vision angle of 15°. To get a proper coverage of the quadcopter's surroundings eight sensors are applied horizontally as can be seen in Figure 1.3. For determining the altitude a ninth sensor located under the frame and a tenth is attached on top for monitoring the distance to the ceiling.

With $8 \cdot 15^\circ = 120^\circ = \frac{1}{3} \cdot 360^\circ$, a third of the environment can be detected reliably, which is sufficient for the case study.

4.3 Outlier Detection and Signal Smoothing

A typical problem with real-time applications are measurement anomalies such as unstable values and outliers. If the sensor value production and processing is fast enough, single wrong values might not affect the system noticeably. With a slow value production like it comes with ultrasonic measurement a wrong value is used for many ticks and influences the behavior of the system determinatively. To prevent this we need to make sure that only correct and smoothened values are processed.

We can divide value errors into two categories, single errors and permanent errors. Single errors such as outliers are particular wrong errors which can occur randomly for several reasons and cannot be prevented with a correct sensor setup. They can be filtered

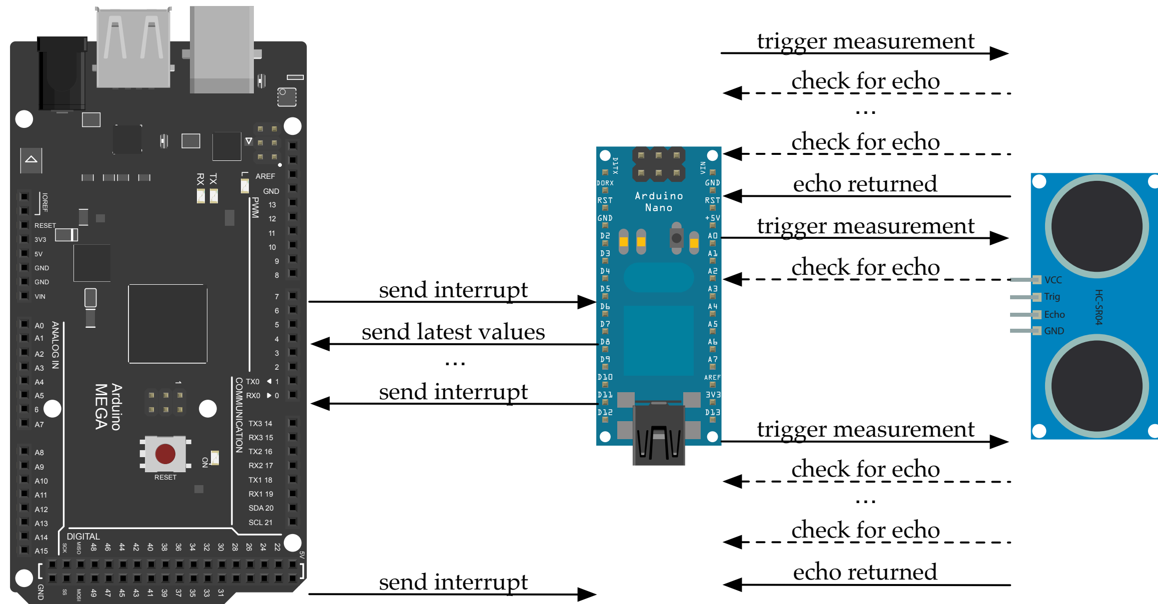


Figure 4.5. Sensor communication between the Arduino Mega and an ulteasonic sensor

out with knowledge about the predecessors and successors. Permanent wrong values from vibrations of the sensors cannot be filtered and need to be prevented by a correct mechanical sensor attachment.

An outlier filter needs to find and smoothen an individual wrong value in a series of correct values. It is important that the filter does not smoothen a strong variance in the measured results caused by an actual movement. Hereinafter four filtering methods are compared.

1. Maximum deviation filter:
   A Maximum deviation filter ignores all measured distances that deviate more than a set value d from its predecessor.

2. Average filter:
   An average filter has a window of defined and preferably odd size w and calculates the average of w values in a row, with $w \in \mathbb{N} > 0$:
   $v_i = \frac{1}{w} \sum_{i-\lceil (w-1)/2 \rceil}^{i+\lfloor (w-1)/2 \rfloor} v_i$
   If the series' borders are reached, the first respectively last values are used multiple times.

3. Weighted filter:
   A weighted filter works similar to the average filter but weights values more the newer

they are. It has a window of defined size w and uses a certain value v_i and its $w-1$ predecessors, with $i, w \in \mathbb{N}, w > 0$:
$v_i = 0.5^{w-1} \cdot v_{i-(w-1)} + \sum_{k=0}^{w-2} 0.5^{k+1} \cdot v_{i-k}$

4. Median filter:
A median filter has defined and preferably odd window of size w and overwrites a value v_i with the median value of all values between $v_{i-\lceil (w-1)/2 \rceil}$ and $v_{i+\lfloor (w-1)/2 \rfloor}$.

When specifying the window size for filters using successors of a value we need to consider that our filter has to be a sliding filter and that with a larger window size the delay between receiving and processing a value gets longer. As it can be seen in Figure 4.6 although the newest value 46 has already been detected due to filtering issues value 44 is currently processed and put out. It will take two more incoming values to have value 46 processed. Because acoustic measurement is relatively slow compared to optical measurement we will use a window size of three to have a filter impact but the delay short.

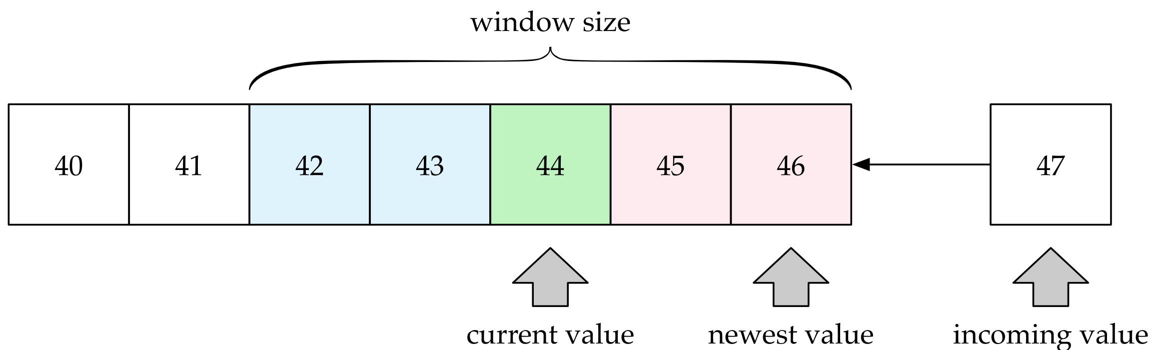


Figure 4.6. Window size delays value processing

To evaluate different filtering techniques we create three generic distance value series to simulate various situations:

Table 4.4 handles a series of nearly constant values with a single outlier due to a measurement error with the different filters mentioned above. Red numbers in the filtered series mark erroneously filtered values. Both the maximum deviation filter and the median filter return the same result and detect the 255 as error. The average filter does not provide an acceptable result. It weakens the deflection but also broadens it. Also the weighted filter can soften the outlier but only slowly get back to the constant values.

With a single outlier within falling values the result is similar as Table 4.5 shows. At this point a difference between the results of the maximum deviation and the median filter ist visible. The maximum deviation filter provides a pure copy of the original unfiltered values by just cutting out the outlier while the median filter causes changes to the values.

Table 4.4. Single outlier in constant value series

Filtering method	Resulting values series							
No filtering	040	042	042	040	041	255	041	040
Maximum deviation	040	042	042	040	041	041	041	040
Average filter	041	041	041	041	112	112	112	040
Weighted filter	040	041	042	041	041	148	095	094
Median filter	040	042	042	041	041	041	041	040

Table 4.5. Single outlier within falling values

Filtering method	Resulting values series							
No filtering	080	070	052	040	198	020	021	020
Maximum deviation	080	070	052	040	040	020	021	020
Average filter	080	075	064	051	122	070	065	020
Weighted filter	077	067	054	097	086	080	020	020
Median filter	080	080	052	052	040	020	020	020

Table 4.6. Quickly falling values without an outlier

Filtering method	Resulting values series							
No filtering	180	182	090	040	080	160	180	181
Maximum deviation	180	182	182	182	...			
Average filter	180	181	136	088	073	110	150	176
Weighted filter	181	151	104	070	093	140	174	181
Median filter	180	180	090	080	080	160	180	181

The last example is the simulation of a quick movement of the quadcopter or the obstacle and a resulting quick change of the measured distances, as it could appear when getting a close object out of sight. Table 4.6 again outlines again problems with the average and the weighted filter, especially showing a great problem with the maximum deviation filter. Depending on the threshold value d in case of a rapid decrease of distance the filter will stay at the last value before the drop. Only the median filter can provide a good result. Because the value 40 was existent for only one sensor cycle, it was smoothened out.

As mentioned above a reason for measurement errors can be the mechanical attachment of the sensors. While operating the quadcopter's motors vibrate in high frequencies, which are transmitted to the sensors as well. The high frequency vibrations disturb the sensors and lead them to recognize their own vibration as vibration of the air, which is interpreted as an acoustic wave.

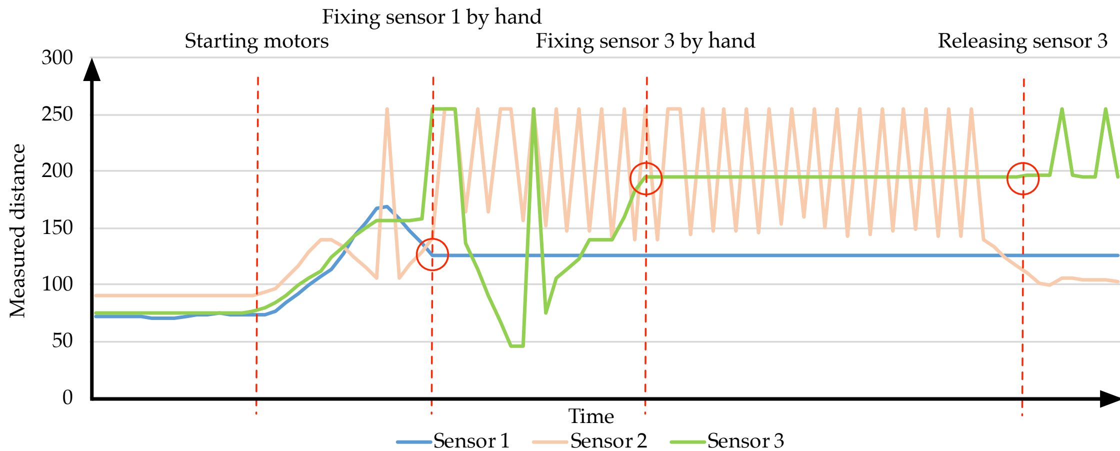


Figure 4.7. Test data from vibrating sensors

Figure 4.7 shows the proximity data of three different sensors over time. Before the first dashed line, the motors are not running and the sensor values are stable. After starting the motors it is visible that the sensor values begin to randomly fluctuate. This shows the influence of the motors' vibrations on the ultrasonic measurements. At the second dashed line sensor 1 is held tightly with two fingers to stop the vibrations of the sensor. Immediately the signal is constant again. The same applies to sensor 3 at the third dashed line. When releasing sensor 3 again at the fourth dashed line the measured values start to fluctuate again.

To solve this problem the sensors were unscrewed and attached with a rubber band and a piece of sponge between them and the quadcopter. With this setup no changes in the sensor data are visible when running the motors.

4.4 Collision Avoidance

After ensuring the ability to get proper proximity values of all sensors as described in the previous sections, rules for forbidden flight directions can be set. For that purpose we define some symbols:

Let

n be the number of horizontally aligned sensors with $n := 8$.

N be the set of natural numbers from 0 to n-1 with $N := \{0, \ldots, n-1\}$.

s_i be the sensor with index $i \in N$. The indices are assigned clockwise beginning with the front sensor.

S be the the set of all horizontally aligned sensors with $S := \{s_i | i \in N\}$.

d_{s_i} be the pointing direction of s_i.

D be the set of the pointing directions of all horizontally aligned sensors with $D := \{d_{s_i} | i \in N\}$.

A_{s_i} be the set of allowed flight directions ascertained by s_i with $A_{s_i} := \{d_{s_i} | d_{s_i} \in D$ is allowed flight direction$\}$. $A^a_{s_i}$ additionally names the set of allowed flight directions with an obstacle under attention threshold and $A^d_{s_i}$ with an obstacle under danger threshold.

F_{s_i} be the equivalent set of forbidden flight directions with $F_{s_i} := \{d_{s_i} | d_{s_i} \in D$ is forbidden flight direction$\}$. $F^a_{s_i}$ and $F^d_{s_i}$ are defined analogously.

Bouabdallah et al. [BBP+07] suggest a three state avoidance procedure for any of the sensors, which is the base for the further considerations. Whenever a sensor does not detect any object closer than the attention threshold it is in the first, uncritical state, called *Okay* in the following. *Okay* means that the quadcopter may move in the direction the corresponding sensor is pointing.

When detecting an obstacle within the attention threshold the regarding sensor immediately switches to the so called *Attention* state. Having one sensor in this state the quadcopter should start moving away from the obstacle but is still controllable in all directions not pointing towards it. The forbidden flight directions for the mentioned sensor s_i are:

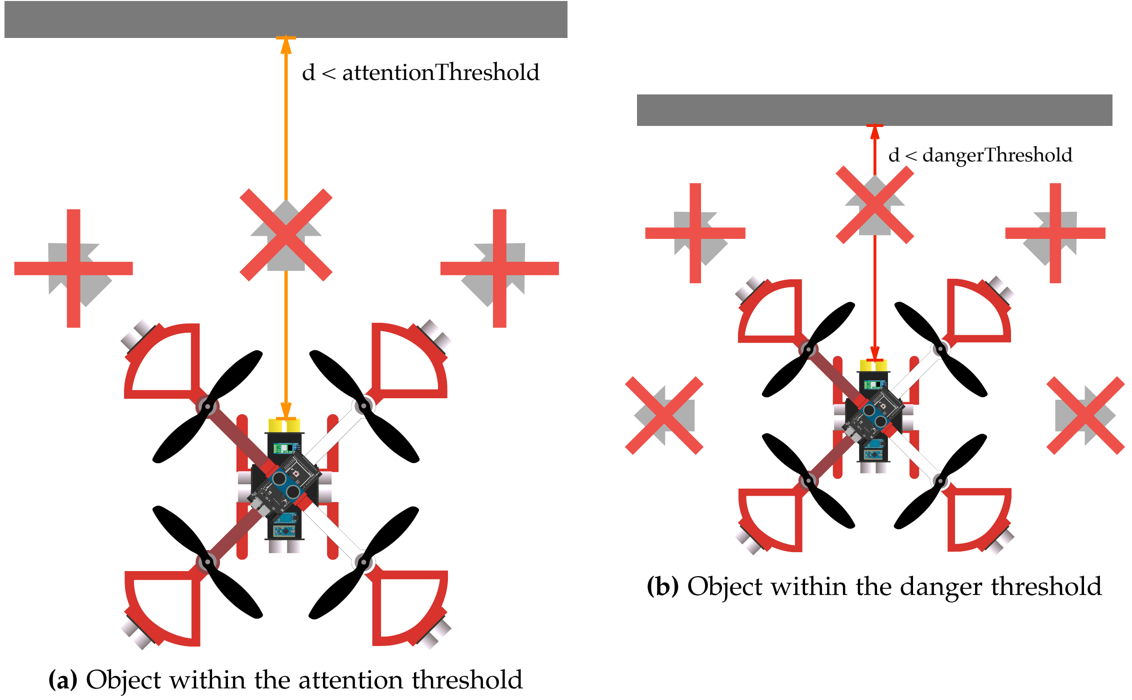
$$F^a_{s_i} = \{d_{s_{(i+7)\%8}},\ d_{s_i},\ d_{s_{(i+9)\%8}}\} \tag{4.4.1}$$

In case any sensor measures a proximity even smaller than the danger threshold, it switches to state *Danger*. *Danger* means getting away from the detected object as quickly as possible. If any sensor is in the state *Danger* the quadcopter is so close to an obstacle that the remote controls are overwritten for any direction not pointing away from it. The forbidden flight directions result as following:

$$F^d_{s_i} = \{d_{s_{(i+6)\%8}},\ d_{s_{(i+7)\%8}},\ d_{s_i},\ d_{s_{(i+9)\%8}},\ d_{s_{(i+10)\%8}}\} \tag{4.4.2}$$

Figure 4.8 demonstrates the resulting forbidden flight directions marked with a crossed arrow, in case the yellow colored sensor detects an obstacle. Depending on the distance d either the first neighbor directions are blocked as shown in Figure 4.8a or also the second ones as visible in Figure 4.8b.

After having the forbidden flight directions that result from a single sensor, the next step is to give thoughts to situations where multiple sensors recognize objects.



(a) Object within the attention threshold

(b) Object within the danger threshold

Figure 4.8. Different obstacle distances lead to different forbidden flight directions, marked with a crossed arrow.

The forbidden flight directions for the whole quadcopter result from the union of the particular sensors' forbidden flight directions:

$$F = \bigcup_{i \in N} F_{s_i} \tag{4.4.3}$$

The set of allowed flight directions is the relative complement of F in D:

$$A = D \backslash F \tag{4.4.4}$$

A helps to verify movement instructions both by humans or an autonomous flight control. Let I^c be a set of instructed flight directions coming from the controller. The interSection of A and I^c produces the set of verified, allowed instructed flight directions:

$$I^{cA} := A \cap I^c \tag{4.4.5}$$

The next step is to make the quadcopter move away from potential obstacles and combine these movements with the ones instructed by the controller I^c. Let therefore I^p be the set of instructed flight directions by the collision avoidance procedure and $op : S \rightarrow S,\ s_i \mapsto s_{(i+4)\%8}$ be a function giving the opposite sensor of the sensor s_i.

Assuming that the quadcopter needs to avoid collisions with any obstacle and hence move away from every alerting sensor's direction, we can get I^p out of F:

$$I^p := \{d_{op(s_i)} | d_{s_i} \in F\} \tag{4.4.6}$$

Again, with help of A we can verify the resulting avoidance procedure instructed movements:

$$I^{pA} := A \cap I^p \tag{4.4.7}$$

As a last step the verified movement instructions I^{cA} and I^{pA} need to be combined to the final set of movements of the quadcopter M:

$$M := I^{cA} \cup I^{pA} = (I^c \cup I^p) \cap A \tag{4.4.8}$$

M is the set of movements that were intended by the controlling instance and that are required to get away from obstacles, verified with the allowed flight directions.

4.5 Error Handling and Crash Avoidance

While avoiding collisions generally means moving away from potential obstacles, it includes some error handling and crash avoidance. In case of a remote connection loss, a system hang-up, or an endless loop resulting from a programming error the quadcopter would not be controllable. This would possibly end in a fatal crash and a partly damaged quadcopter. To understand the risk of a connection loss some knowledge about the functionality of the copter's remote control is needed.

The quadcopter's flight commands consist of accelerating and slowing down the motors for height control, turning the quadcopter around the z-axis in both directions and tilting it on the x- and y-axes. Each command increases or decreases a set value, which is for turning and tilting an angle. Whenever a command is sent to quadcopter the flight controller adjusts the desired angles of all axes and tries to reach them. In case of a connection loss with a set tilt angle unequal zero the quadcopter stays at it orientation and starts drifting to a direction and cannot be caught again. For this reason it is mandatory that the flight controller recognizes a connection error. Due to many error sources the most reliable way is to send a keep-alive signal from the ground station. As long as the flight controller frequently receives the keep-alive signal the connection is okay. If after a specified time no keep-alive signal has arrived the quadcopter initiates an emergency landing. Test have shown that a transmission interval of 300ms is adequate for not overloading the Bluetooth connection, which sends and receives alternately, and being quick enough to land in case of an error before a crash.

Preventing a crash caused by a hang-up or endless loop in the flight controller, in which case no command could be processed with the same result as outlined above, is not doable on software side by the microcontroller alone. The only way is to reset the Arduino and

accept a fall to the ground to prevent harm to people and other objects. While a similar keep-alive signal between one of the additional Arduinos and the main Arduino could be established to initiate a direct reset if the flight controller does not respond, an easier and safer way is the included Watchdog on the Arduino Mega 2560. Watchdog can be described as a hardware side counter which, if it reaches a specific value, resets the microcontroller automatically. To prevent it from being restarted the software on the Arduino must reset the timer frequently before it runs full.

Chapter 5

Implementation

Below, the implementation of the developed software components is described in detail. This includes the SCCharts and C/C++ realizations of the collision avoidance as well as the framework on the Arduino. Also the software running on the sensor-processing Arduinos and the telemetry software are presented.

5.1 Host Code Framework

Running an SCChart in an Arduino sketch requires some host code around. Like every Arduino project, our code has a main and executed file named the same as the project. This main file has an initially called *setup* function and a repeatedly triggered *loop*. Although this is very similar to the SCCharts' generated C code, host code operation need to be performed before the actual collision avoidance can work. For this reason a framework of host code was written in C/C++ on the Arduino, which can invoke either the C/C++ implementation of the collision avoidance or the SCCharts version. Listing 5.1 provides an shortened overview about the important parts of this framework. Some functions are combined and some are renamed for better understandability. All bold printed functions are important for this thesis.

As described in Section 4.5, the Watchdog timer can be enabled in the *setup* function as visible in line 7 and reset in every *loop* in line 12. *proximityRead* in line 14 and *proximityCheck* in line 20 are covered later in Section 5.3. Section 4.5 also mentions the use of a keep-alive signal to initiate an emergency landing if the remote connection fails. Lines 22 to 24 handle the monitoring of the keep-alive signal in every tick.

```
1  void setup() {
2    initiateSerialCommunication(); // via the Bluetooth connection
3    initializeInterruptsAndCommunicationForAditionalArduinos();
4    initializeGyroscope();
5    initializePIDValues(); // for the stabilization routine
6    wdt_enable(WDTO_60MS); // Enable watchdog timer
7  }
8
9  void loop() {
10   wdt_reset(); // Reset watchdog timer
11   readGyroscope();
```

```
12    proximityRead(); // Read new proximity values if available
13    if (motionTrackingChipReady && !setupDone)
14      setupDone = !setupDone;
15    if setupDone {
16      processIncomingCommands();
17      proximityCheck(); // Process latest proximity values
18      recalculateAngles();
19      if (currentTime − lastKeepAlive >= keepAliveInterval)
20        emergencyLanding();
21      recalculateMotorValues();
22      writeDebug();
23    }
24  }
```

Listing 5.1. *loop* function of the ultrasonic sensor controller

5.2 Ultrasonic Sensor Controller

In Chapter 4.2 the fact, that the handling of all ten sensors requires two separate Arduinos, was discussed. On each of these one instance of the ultrasonic sensor controller is executed. This controller makes use of the afore-mentioned NewPing Library for Arduino.

```
 1 void loop() {
 2   for (uint8_t i = 0; i < SONAR_NUM; i++) {
 3     if (millis() >= pingTimer[i]) {
 4       pingTimer[i] += PING_INTERVAL * SONAR_NUM;
 5       if (i == 0 && currentSensor == SONAR_NUM − 1) sensorCycle();
 6       sonar[currentSensor].timer_stop();
 7       currentSensor = i;
 8       for(uint8_t j = 0; j < ARRAY_LENGTH − 1; j++)
 9         cm[currentSensor][j] = cm[currentSensor][j+1];
10       knownDistances[currentSensor] = false;
11       sonar[currentSensor].ping_timer(echoCheck);
12 } } }
```

Listing 5.2. *loop* function of the ultrasonic sensor controller

Listing 5.2 displays the controllers *loop* function, as it is recommended by the library's developer with a little modification. Every call checks for every sensor if its pingTimer is reached. If this is the case, the respective timer is increased by *PING_INTERVAL* times the number of sensors in line 4. If the most recently checked sensor was the sensor with index 9, the *sensorCycle* function is called. Afterwards, the respective sensor timer is stopped in line 6 and again started in line 11. *ping_timer(echoCheck)* triggers the sensor in the *sonar* array and starts calling *echoCheck* every 24µs.

New in this implementation are lines 8 to 10. Line 10 is a boolean array for all sensors, which specifies whethter any sensor's echo has returned or is missing. Lines 8 and 9 shift the array of the most recently detected distances left before the new distance arrives.

```
1  void sensorCycle() {
2    if(active) {
3      active = false;
4      for (uint8_t i = 0; i < SONAR_NUM; i++) {
5        if(knownDistances[i])
6          Serial.write(max(min(cm[i][0],cm[i][1]),min(max(cm[i][0],cm[i][1]),
             cm[i][2])));
7        else
8          Serial.write((uint8_t) 255);
9      }
10     delayMicroseconds(500);
11     digitalWrite(INTERRUPT_SEND, HIGH);
12     delayMicroseconds(10);
13     digitalWrite(INTERRUPT_SEND, LOW);
14 } }
```

Listing 5.3. *sensorCycle* function of the ultrasonic sensor controller

The *oneSensorCyle* function checks whether the flight controller is ready to receive new sensor data and has sent an interrupt which set *active* to *true* in the interrupt service routine not shown in the listings. Lines 4 to 9 in Listing 5.3 shows that if this happens, the controller sends the median of the latest three received values, in case *kownDistances* is true for the sensor, or the default value for "out of range" 255 to the flight controller for each sensor. After all sensors are processed, lines 10 to 13 send an interrupt back to the main Arduino.

When *echoCheck* is called, *check_timer* in line 2 of Listing 5.4 checks if the sensor's echo has returned within the preset distance's time. If it has, the respective field in the *knownDistances* array is set to *true* and the latest value for the sensor is updated.

```
1  void echoCheck() {
2    if (sonar[currentSensor].check_timer()) {
3      knownDistances[currentSensor] = true;
4      cm[currentSensor][ARRAY_LENGTH - 1] = (sonar[currentSensor].ping_result
           / US_ROUNDTRIP_CM);
5  } }
```

Listing 5.4. *echoCheck* function of the ultrasonic sensor controller

5.3 Collision Avoidance in C/C++

The first implementation is done in C/C++ directly on the Arduino. Although the SCCharts language has been in development for a few years, testing and debugging with an Arduino is not as intuitive as in pure C/C++. A non SCCharts version of the software also helps comparing the implementation process.

For the further understanding some information about the proximity value reading is given. Every tick calls the *proximityRead* function, which checks whether one of the sensor-handling Arduinos has sent new data. Listing 5.5 gives an extract for one of the Arduinos and shows the procedure.

```
1  if (newData[i]) {
2    newData1 = false;
3    newProxData++;
4    for (int i = 0; i < 5; i++) {
5      if (Serial2.available() > 0) {
6        cm[i] = Serial2.read();
7      }
8    }
9    digitalWrite(INTERRUPT_SEND1, HIGH);
10   delayMicroseconds(10);
11   digitalWrite(INTERRUPT_SEND1, LOW);
12 }
```

Listing 5.5. Extract of the *proximityRead* function for one Arduino

Whenever the *proximityRead* function finds the according field in *newData* on *true*, the first additional Arduino has sent new values and triggered the first interrupt service routine. As a result the variable is set to *false* immediately, and the counter *newProxData* is incremented. *newProxData* helps the flight controller to know when all additional Arduinos have sent their values. The loop from line 4 to line 8 in Listing 5.5 reads all received values on the Serial port, which is connected to the respective Arduino, and writes them to the *cm* array. Subsequently, an interrupt is sent in lines 9 to 11 to notify the sensor-handling Arduino that all values have been received and the flight controller is ready to get new values during the next tick.

Because in Arduino the standard C++ library is missing[1], containers such as sets cannot be used by default. As SCCharts does also not support such objects, the collision avoidance algorithm implementation is done without a third party library or the usage of sets by creating an array of allowed flight directions directly as described in Section 4.4.

Before the first step of determining the forbidden and allowed flight directions using the sensors' proximity values, the *proximityCheck* function defines the *attentionThreshold*,

[1]https://www.arduino.cc/en/Reference/Libraries

the *dangerThreshold*, and the *strength*, which declares to which degree the quadcopter needs to be tilted away from a detected obstacle in case of a potential collision.

```
1 void proximityCheck() {
2   int attentionThreshold = 130;
3   int dangerThreshold = 80;
4   int strength = 2;
```

Listing 5.6. Variable declaration and definition in the *proximityCheck* function

Next, an array of size 8 for the allowed and forbidden directions *allowedDirections* is created in line 5 in Listing 5.7. Every field is initiated with -1 to indicate that is has not been set yet. Each array field is reserved for a direction with indices corresponding to the indices of the sensors as described in Section 4.4. If any field later holds the value of 0 this means a blocked direction, if it holds the value of 1 the direction is allowed. To fill the array with the according values, a loop is run over all sensors to check whether any of them has detected an obstacle in a distance below the attentionThreshold. If this check evaluates to *true* for a sensor a *range* variable is set to 1 in line 8. *range* defines the number of neighbor directions on both sides, that are blocked as well, as shown in Figure 4.8. In case the measured distance is even smaller than the *dangerThreshold*, *range* is set to 2.

With Equation (4.4.3) and Equation (4.4.4) the allowed flight directions are the directions, which are not forbidden by any sensor. This means that any sensor can overwrite an allowed flight direction but not vice versa. For this reason, if a sensor detects an object too close and the *range* is set to 1 or 2, in line 12 the respective sensor's and its neighbors' allowed directions can be set to 0 without further checks.

If no obstacle is detected within the attention radius the according field in the *allowedDirections* array can be set to 1 if and only if the field has not been changed from -1. If any other sensor has already blocked the direction, another change of the field is not permitted.

```
5   int8_t allowedDirections[8] = {-1, -1, -1, -1, -1, -1, -1, -1};
6   for(int8_t i = 0; i < 8; i++) {
7       if(cm[i] < attentionThreshold) {
8         int8_t range = 1;
9         if(cm[i] < dangerThreshold)
10          range = 2;
11        for(int8_t j = i-range; j <= i+range; j++)
12          allowedDirections[(j+8) % 8] = 0;
13      } else {
14        if(allowedDirections[i] < 0)
15          allowedDirections[i] = 1;
16  }   }
```

Listing 5.7. Creating the *allowedDirections* array in the *proximityCheck* function.

After having an array of allowed flight directions from all sensors, the second step is to extend the controller-initiated flight directions by adding movements to escape obstacle collisions. The current set values for tilting the quadcopter on the x- and y-axes are named *setX* and *setY*. For every forbidden flight direction it the algorithm checks whether the present *setX* and *setY* values are tilting the quadcopter more than the angle of *strength* away from it. As line 17 in Listing 5.8 shows, in case of an obstacle in front of the sensor with index 0, it requires a negative *setY* value to avoid a collision. If the controller-intended *setY* value is already smaller than a negative *strength* it does not need to be changed. In case it is not, it will be overwritten.

```
17  if(allowedDirections[0] == 0) setY = (setY<-strength) ? setY : -strength;
18  if(allowedDirections[4] == 0) setY = (setY>strength) ? setY : strength;
19  if(allowedDirections[2] == 0) setX = (setX<-strength) ? setX : -strength;
20  if(allowedDirections[6] == 0) setX = (setX>strength) ? setX : strength;
21  if(allowedDirections[1] == 0) {
22     setY = (setY<-strength) ? setY : -strength;
23     setX = (setX<-strength) ? setX : -strength;
24  }
25  if(allowedDirections[3] == 0) {
26     setY = (setY>strength) ? setY : strength;
27     setX = (setX<-strength) ? setX : -strength;
28  }
29  if(allowedDirections[5] == 0) {
30     setY = (setY>strength) ? setY : strength;
31     setX = (setX>strength) ? setX : strength;
32  }
33  if(allowedDirections[7] == 0) {
34     setY = (setY<-strength) ? setY : -strength;
35     setX = (setX>strength) ? setX : strength;
36  }
```

Listing 5.8. Expanding the controller-intended movements by movements to escape obstacles.

In some situations it can happen that both the front sensor and the back sensor recognize an object within the attention radius. In this instance line 17 would verify that the *setY* value is *-strength* or smaller and line 18 would overwrite *setY* with *strength*. To solve this problem the resulting *setX* and *setY* values are verified with the earlier created *allowedDirections* array. Equation (4.4.8) explains the composition of the final quadcopter's movements. It takes the combined movements intersected with the allowed flight directions. Since a positive *setY* value tilts the quadcopter forwards it needs to be accompanied by the allowed flight directions front-left, front, and front-right. Similarly, all tilting directions can be verified, as shown in Listing 5.9.

```
37    if(setY > 0) {
38      setY = (allowedDirections[-1]
39           && allowedDirections[0]
40           && allowedDirections[1]) ? setY : 0;
41    } else {
42      setY = (allowedDirections[3]
43           && allowedDirections[4]
44           && allowedDirections[5]) ? setY : 0;
45    }
46    if(setX > 0) {
47      setX = (allowedDirections[1]
48           && allowedDirections[2]
49           && allowedDirections[3]) ? setX : 0;
50    } else {
51      setX = (allowedDirections[5]
52           && allowedDirections[6]
53           && allowedDirections[7]) ? setX : 0;
54    }
```

Listing 5.9. Verifying the allowed directions.

5.4 Collision Avoidance in SCCharts

Implementing the presented collision avoidance procedure in the synchronous language SCCharts can be done similarly. Figure 5.1 illustrates the variable initialization and all three mentioned steps in separate superstates, which are connected via immediate termination transitions.

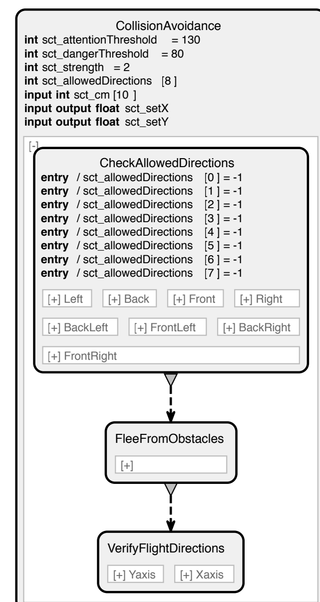


Figure 5.1. Collapsed SC-Chart

It also shows how the first superstate *CheckAllowedDirections* initializes the *allowedDirections* array with -1 and includes eight regions with threshold checks for one of the horizontally aligned sensors each. The initial state in every region is *Unknown*, as can be seen in Figure 5.2, which must lead to one of the following states immediately. In case the attention threshold or even both thresholds are undercut it is switched to state *Attention* or *Danger* and the sensor's direction as well as its regarding neighbors' directions are forbidden. If no obstacle for this sensor is within the threshold radius the SCChart ends with state *Okay* and if and only if the sensor's direction has not been modified by another sensor, the direction is allowed. The "initialize-update-read" protocol of SCCharts ensures that the parallel states in the *CheckAllowed-*

Directions superstate are handled correctly and all *Okay* states' entry actions are executed after the ones of the states *Danger* and *Attention*.

SCCharts provides the ability of using referenced states. With this technique and bound variables a second SCChart can be integrated, if necessary multiple times, in an existing region. Listing 5.10 shows how the second SCChart *Sensor* can be referenced as a state. Every region contains an *index* integer that is passed to the referenced SCChart in addition to other required variables.

```
1 region Front:
2   int index = 0;
3   initial final state ThresholdCheck references Sensor
4   bind index to index,
5        attentionThreshold to attentionThreshold,
6        dangerThreshold to dangerThreshold,
7        cm to cm,
8        allowedDirections to allowedDirections;
```

Listing 5.10. Region definition for referenced threshold check SCChart

While there exist some depiction errors with the entry actions, it is visible in Figure 5.3 how the bound *index* variable is used to identify the regarding fields in the *cm* and *allowedDirections* array. Unfortunately using variables as array identifiers is currently not supported, which is why the implementation is realized as described in Section 5.4 without referenced states.

The second state *FleeFromObstacles* is responsible for the movement extension. As shown in Figure 5.4, it includes the sequential check for all directions. Analogously to the second step in Section 4.4, every field in the *allowedDirections* array is checked, and if it's direction is forbidden, the *setX* or *setY* values are adjusted until the termination transition leads to the last superstate *VerifyFlightDirections*.

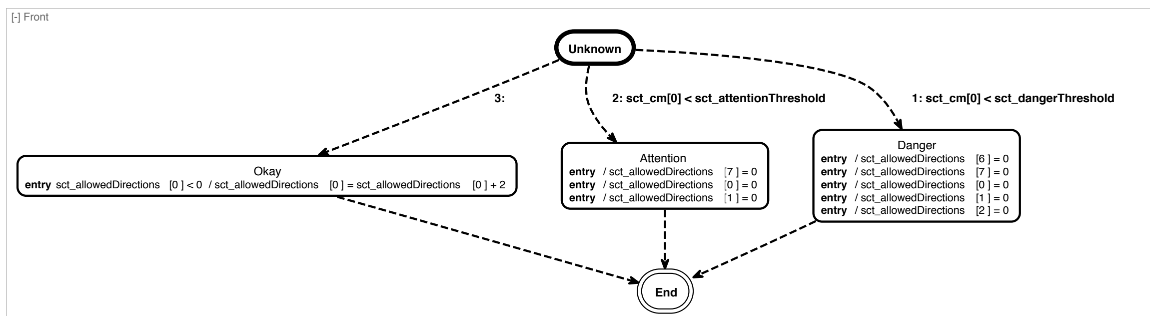


Figure 5.2. Referenced sensor SCChart

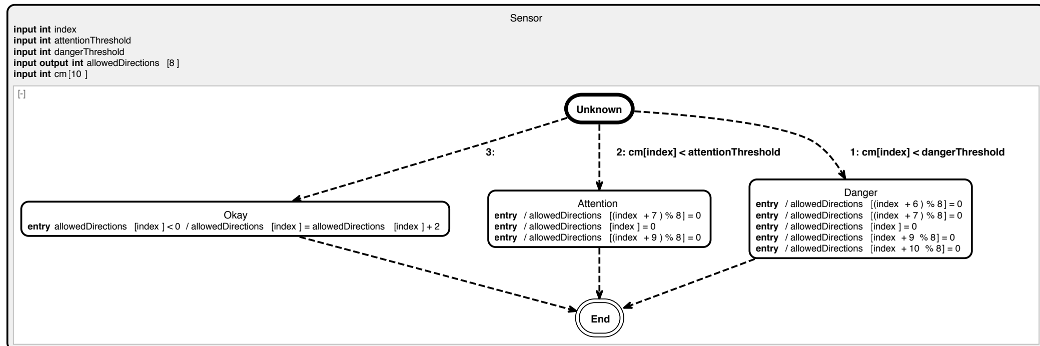


Figure 5.3. Referenced sensor SCChart

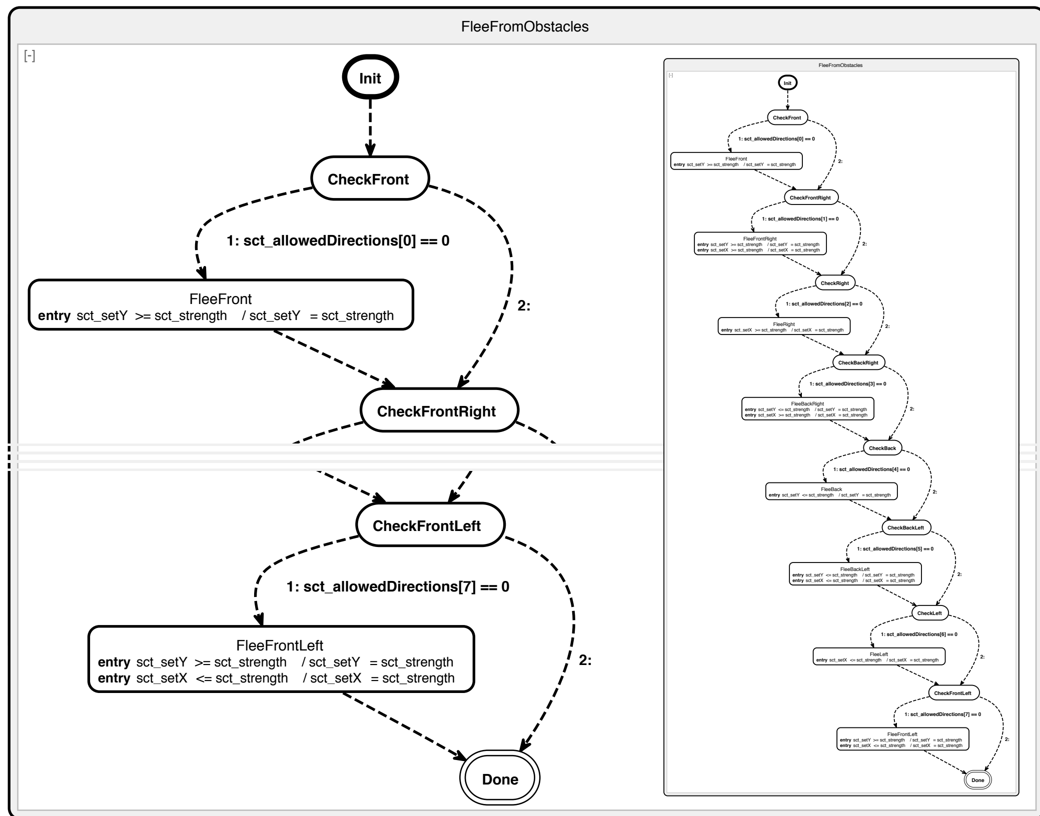


Figure 5.4. Shortened movement extension in the SCChart with small preview of the whole superstate on the right

As described before, both set values are verified in the last step *VerifyFlightDirections*. Because the only writing operations are the adjustment of both values, they can be processed in two separate regions. Analogously to the last described step in Section, Figure 5.5 presents the final movement verification.

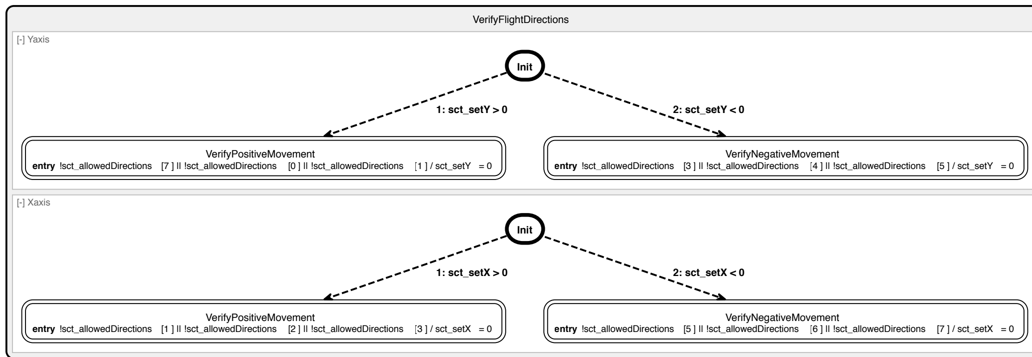


Figure 5.5. Movement verification in the SCChart divided into two regions

The complete SCChart is called in every tick in which at least two ultrasonic sensor-handling Arduinos have sent new values by the flight controller. Hence, all transitions must be immediate to process the new values instantaneously.

5.5 Telemetry Software

For the remote control and to receive responses from the quadcopter, a telemetry software in Java 8 and JavaFX was written to provide compatibility to all common operating systems. In addition, the jSerialComm library introduced in Section 3.3 gives an environment independent access to serial ports.

The interface is structured into three parts, as visible in Figure 5.6. The very top of the window holds control elements and information icons for the connection to and status of the quadcopter. Listing A.1 gives a code snippet which shows how easily the jSerialComm library can be used.

The second segment consists of a set of buttons for controlling the movements and leaves space for other toggles of different usages. As long as the window is focused the quadcopter can also be controlled via keyboard inputs. The exact assignment of keys can be found in Appendix B.

Below the controls a log view is located, which shows every pressed key and every initiated action as well as incoming messages from the quadcopter with the regarding timestamps. This way, in case of a critical incident the course of events can be comprehended. To receive messages after a successful connection a separate thread is started to open an input stream and catch all incoming data.

As mentioned in Section 4.5 the flight control of the quadcopter needs a periodical keep-alive signal to be sure that the Bluetooth connection is okay. Therefore the telemetry software starts a *keepAliveHandler* together with the motors. This handler sends the keep-alive signal *0x99* every 300ms.

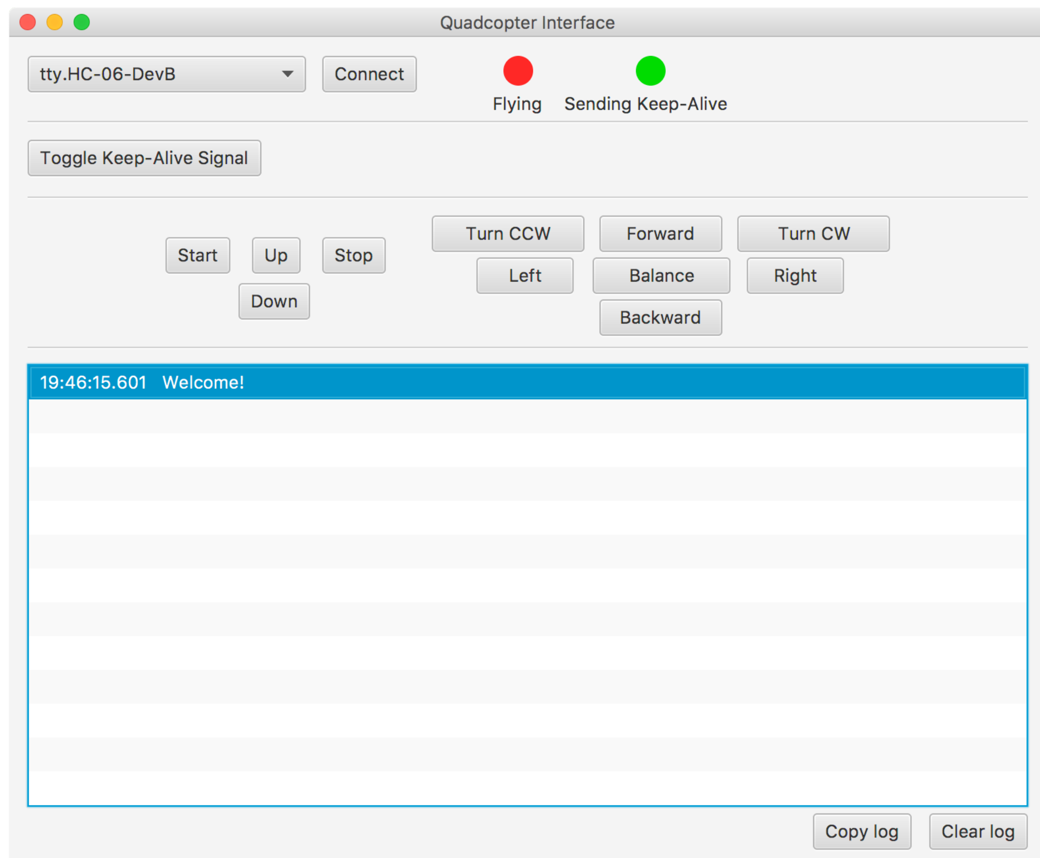


Figure 5.6. Telemetry Software

Chapter 6

Evaluation

The evaluation is divided into two sections, the judgement of the sensor behavior and the comparison of both software solutions. For the sensor processing it is important to assess the quality of the values that are used by the collision avoidance algorithm. Comparing both software solutions helps to find problems with SCCharts but also shows its strengths.

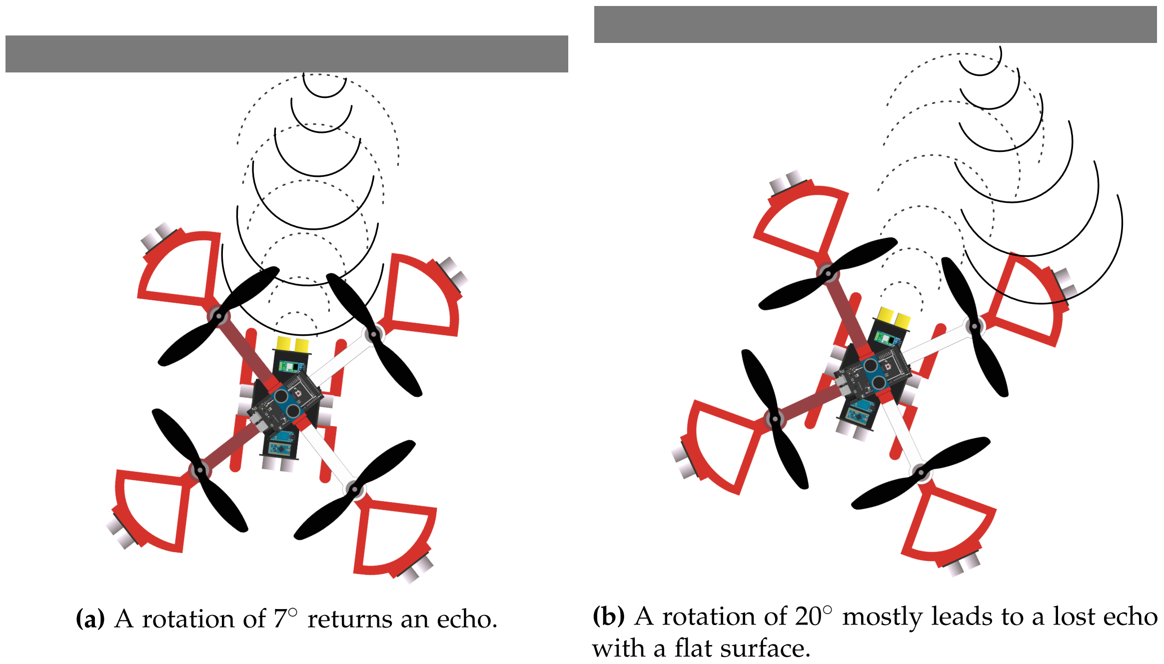
6.1 Sensor Behavior

During flight a UAV is exposed to strong airstreams which make a stable flight complicated. This leads to small unpreventable movements like shaking and vibrating on different axes. To evaluate the value correctness a test person can hold the quadcopter in both hands with a monitored sensor directed to a proper wall and simulate typical flight movements. There are three possible motions that change the sensor's view on the wall:

1. Moving the quadcopter up and down:
   When pointing the sensor towards a flat surface and moving the quadcopter up and down frequently without changing the sensor's view angle, the sensor value stays constant.

2. Tilting or turning the quadcopter:
   While turning the quadcopter on the z-axis or titling it towards the wall and back, the distance logically increases, but a small angle of about 15° is enough to lose the obstacle and get the "out of range" value instead. Section 4.2 explains the reason behind lost echoes. Figure 6.1 sketches the movement and the loss of the sensor's echo.

3. Moving the quadcopter towards the wall and away from it:
   With movements towards the wall or away from it, the correctness of the measured values depends on the movement speed as Figure 6.2 shows. Over a short period of time two approaches of increasing the distance to the wall are made. The orange graph describes the value variation with a quick movement while the blue graph displays a moderate movement. The results can be explained by the Doppler Effect and emerge to be one of the big disadvantages with sonic distance measurement in comparison to measuring with light. As a consequence the quadcopter must not do any quick movements to avoid wrong measurements of this type.



(a) A rotation of 7° returns an echo.

(b) A rotation of 20° mostly leads to a lost echo with a flat surface.

Figure 6.1. Rotating the quadcopter only a few degrees can cause the loss of a sensor's echo.

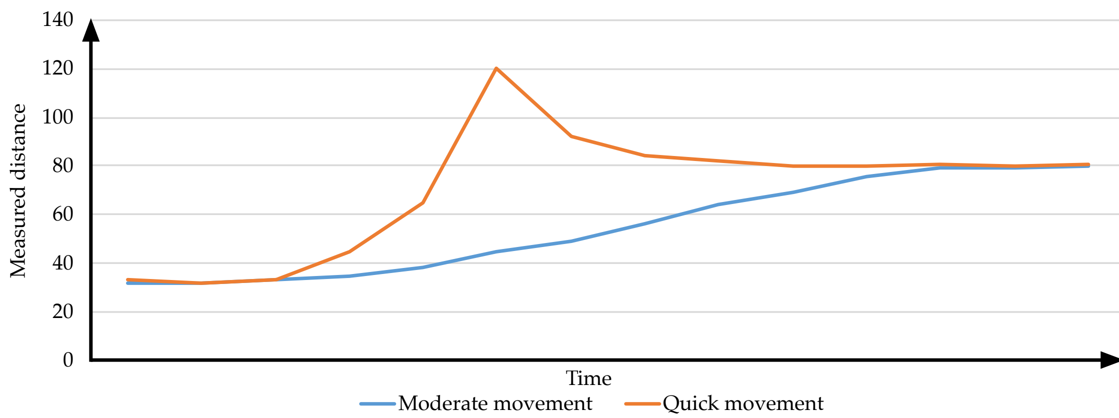


Figure 6.2. Measured values from one sensor while increasing the distance to a wall with different speed.

A similar problem results from the vibrations of the sensors, as already brought up in Section 4.3, which seems to be solved with the pieces of sponge attached between the sensors and the quadcopter.

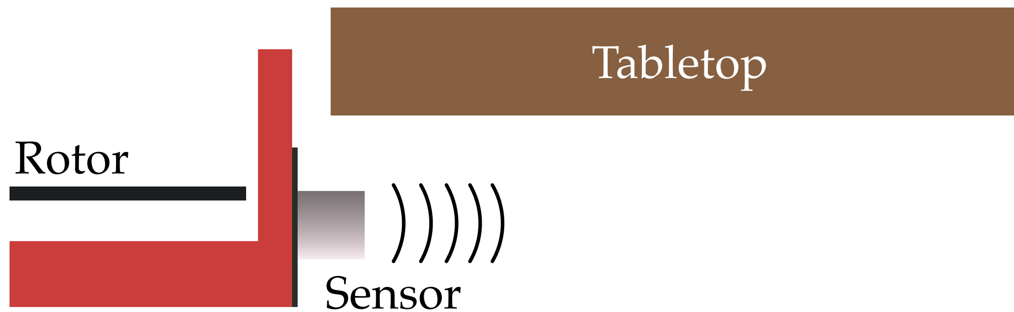


Figure 6.3. Side view on one of the quadcopter's arms next to a tabletop. The sensor does not recognize the table.

When pointing the sensor towards a person wearing clothes or a soft object such as a curtain, the second reason for lost echoes can be experienced, as it was mentioned in Section 4.2. This makes a deployment in a populated area hard.

Even with a large delay between processing the sensors, the in Section 4.2 described ghost echoes can be noticed when testing in a large room with multiple obstacles such as pillars or edges. With more space the lifetimes of sound waves are much longer and round pillars increase the chance of a sensor detecting echoes from other sensors.

The sonic waves created by the sensors propagate horizontally to have a preferably wide but 2-dimensional field of view. In the instance of encountering a horizontally aligned flat object such as a table, the quadcopter might be on the same height but have the sensors underneath the table top. The quadcopter has no chance of detecting the obvious obstacle, as sketched in Figure 6.3.

6.2 Collision Avoidance Implementation

At first both implementations are compared by numbers.

▻ The C/C++ implementation counts 55 lines of code.

▻ The code which was needed to model the SCChart counts 300 lines.

▻ The C code that was compiled from the SCChart counts 745 lines.

The result shows the typing effort to write the implementations by hand. The pure C/C++ code is probably written in less time while the development of the idea cannot be measured and compared here. One implementation needed to be written first and in turn inspires the second one.

Although the C/C++ version is much smaller, both versions can be included in an Arduino sketch and compiled with the related compiler.

▻ The compiled C/C++ implementation is 5.382 bytes in size.

▻ The compiled SCCharts model is 67.032 bytes in size.

This comparison shows the required storage on the device. Both sizes can be handled by an Arduino without difficulty. One reason of the increased file size is the concurrent handling of the sensors in SCCharts instead of the sequential processing in C.

After including the both versions in an Arduino sketch each, the runtime of both algorithms can be compared.

▻ The C/C++ implementation takes 16 microseconds in average.

▻ The SCCharts implementation takes 216 microseconds in average.

In contrast the SCCharts excel by a much higher understandability. Though this particular implementation does not use much of concurrency, the sequential flow is easily visible. This tallies with the survey results of *The Railway Project Report* [SMS+15].

In the following both implementations are compared concerning the realization. The C/C++ version of the first step of creating the array for the allowed flight directions is solved sequentially. That way it is ensured that only one sensor at a time is processed and no race conditions can appear. In case two adjacent sensors would be handled simultaneously the following scenario could occur:

1. Sensor 1 does detect an obstacle within the attention radius.
2. Sensor 2 does not detect an obstacle within the attention radius.
3. Sensor 2 checks whether its direction has been blocked by another sensor, which is not the case.
4. Sensor 1 checks whether its direction has been blocked by another sensor, which is also not the case.
5. Sensor 1 sets its direction and both neighbors' directions to blocked.
6. Sensor 2 does not know that it must not allow its direction and sets it to allowed.

This possibility would lead to problems during flight regarding collision avoidance. In SCCharts the MoC solves that issue. Because of the "initialize-update-read" protocol in concurrent regions, reading states are processed after updating states. At first all sensors with detected obstacles are processed because they do not do any further checks with their field in the *allowedDirections* array. All sensors that do not detect any obstacle within the attention radius check if their array field has been changed and are consequently processed afterwards. This example reveals a strong point of modeling in SCCharts.

The second step of extending the intended movements by adding the escape movements cannot be done concurrently in SCCharts. Listing 5.8 displays that if for example the front sensor and the back sensor were handled simultaneously, both had to read the value of *setY* at first and overwrite it afterwards depending on the evaluation. Since for both sensors this would result in a different *setY*, the "initialize-update-read" protocol did not have a solution for this condition and didn't know which of both sensors to process at first. Within the project a discussion arose about the problem that it is currently not feasible to develop an SCChart for a system such as the quadcopter without writing a framework of host code. Although it would be possible to include the call of an exemplary sensor-processing function in a during action of a superstate, it cannot be told when exactly the call is executed not to mention guaranteeing that the sensor-processing function is the first to be called in every tick.

When using a single tick function for a modeled SCChart with a negligible loop time the timing of one state cycle would be equal no matter if taking one transition per tick with non-immediate transitions or taking all transitions in one tick with immediate transitions. This is different when including the tick function in another one with an execution time t. Let it take 20 states for one hypothetical state cycle with an average state execution time of s. It would take $20 \cdot (t + s)$ to get a result out of the SCChart when using non-immediate transitions but only $20 \cdot s$ when using immediate transitions. Whenever multiple states of an SCChart can be connected via immediate transitions, they should.

Chapter 7

Conclusion

To review the whole work the significant results are listed below. In addition some previously unmentioned points are brought up, which are qualified for future work with relation to this project.

7.1 Summary

This thesis described the creation of a basic collision avoidance system for a UAV in the synchronous language SCCharts. Therefore external positioning and internal proximity sensors were compared. It turned out that external hardware does not meet the assumptions of an autonomous system in an unknown environment. After a comparison of internal proximity sensors ten affordable non-processing HC-SR04 ultrasonic sensors were chosen. Different approaches were declined on how to handle the real-time typical problems of a slow sensor value generation beside the safety-critical operation of stabilization. As a final solution two extra Arduinos with the NewPing Library for Arduino handle the sensor evaluation. In addition, problems with unstable values were best solved by a sensor attachment that does not transfer the vibrations of the motors and by using a sliding median filter.

To avoid collisions based on the resulting sensor data, a mathematical approach on handling allowed and forbidden flight movements was made. Intersecting the combination of the set of movements which are intended by a human or an autonomous flight controller and the set of calculated escape movements from potential obstacles with the set of allowed flight directions led to the resulting actual flight movements.

Avoiding collisions includes dealing with potential errors that could prohibit the flight controller from continue working. The consequences of both a remote connection loss as well as a hangup or endless loop on the microcontroller were examined and caught.

Based on the gathered results the required code for the sensor handling Arduinos and the requested telemetry software for remote control was written. Finally, the results of the collision avoidance concept are implemented in two versions, a pure C/C++ version and an SCCharts version, which are compared afterwards. The evaluation also reviews the whole work and discusses problems that could not be solved.

7.2 Future Work

Although the intended goal was reached and the quadcopter is able to recognize objects in its surroundings, the detection is far from perfect. Increasing the number of sensors would result in a higher resolution and probably a better obstacle detection. Also replacing the sensors or combining the currently used sensors with a different type like IR sensors would likely improve the results. For better value stability a combination of multiple filters is imaginable.

The telemetry software allows sending commands to the quadcopter and receiving messages. Due to the limited capacity of the Bluetooth connection not all debug information can be transmitted at once. To change the output, the software of the quadcopter needs to be adjusted and reinstalled. For debugging reasons it would be very helpful if the particular debug entries could be enabled and disabled remotely via the telemetry software. Alternatively, the communication device could be switched to another technology with more capacity. Another important feature for the telemetry software should be the ability to safe log files. This function is also not implemented at that moment.

If the stability of the quadcopter during flight could be enhanced, extended flight maneuvers for collision avoidance could be established. With the help of the attached sensors it would be possible to determine position and movement speed within a room which could together with the collision avoidance lead to an autonomous navigation between a starting point and an end point for instance.

The actual protection of the quadcopter could be increased by better rotor guards or even a cage-like casing.

Acknowledgements

Finally, I would like to express my sincere gratitude to the following persons:

Prof. Dr. Reinhard von Hanxleden for placing the confidence in us and making it possible to participate in this great project to write this thesis.

Our advisors *Dipl.-Inf. Steven Smyth* and *Dipl.-Inf. Christian Motika* for their effort and time up to the evening.

Arnd Plumhoff for making his 3D printer available and for being on hand with help and advice for us. You saved us dozens of times.

My project partners *Annika Pooch*, *Lars Peiler* and *Lewe Andersen* for being such a good team without any real conflict in the six months of cooperation.

Appendix A

Source Code

```
1  public void connect() {
2        if(!connected) {
3            port = SerialPort.getCommPorts()[portSelection.getSelectionModel().
                 getSelectedIndex()];
4            log("Trying to connect to port " + port.getSystemPortName());
5            if (port.openPort()) {
6                log("Successfully connected to port.");
7                connected = true;
8                connectButton.setText("Trennen");
9                port.setComPortTimeouts(SerialPort.TIMEOUT_READ_SEMI_BLOCKING,
                     0, 0);
10               port.setBaudRate(57600);
11               if(startSerialReader()) {
12                   log("Start listening to the quadcopter.");
13               } else {
14                   log("Could not start listening to the quadcopter.");
15               }
16           } else {
17               log("Unable to open the port. Please make sure the port is not
                     blocked or restart the quadcopter.");
18           }
19       } else {
20           if(stopSerialReader()) {
21               port.closePort();
22               connected = false;
23               connectButton.setText("Verbinden");
24               log("Connection closed.");
25           } else {
26               log("Could not stop listening to the quadcopter. Please try
                     again.");
27           }
28       }
29   }
```

Listing A.1. Opening serial port in the telemetry software.

Appendix B

Flight Control

The following gives an overview about the available keyboard inputs at the point of writing for controlling the quadcopter with the telemetry software or a serial terminal.

q Start motors.

e Initiate emergency landing if not on ground and stop motors afterwards.

w Throttle up.

s Throttle down.

i Tilt forwards.

k Tilt backwards.

j Tilt left.

l Tilt right.

r Balance: set tilting angles to zero.

u Turn ccw.

o Turn cw.

Appendix C

Wiring

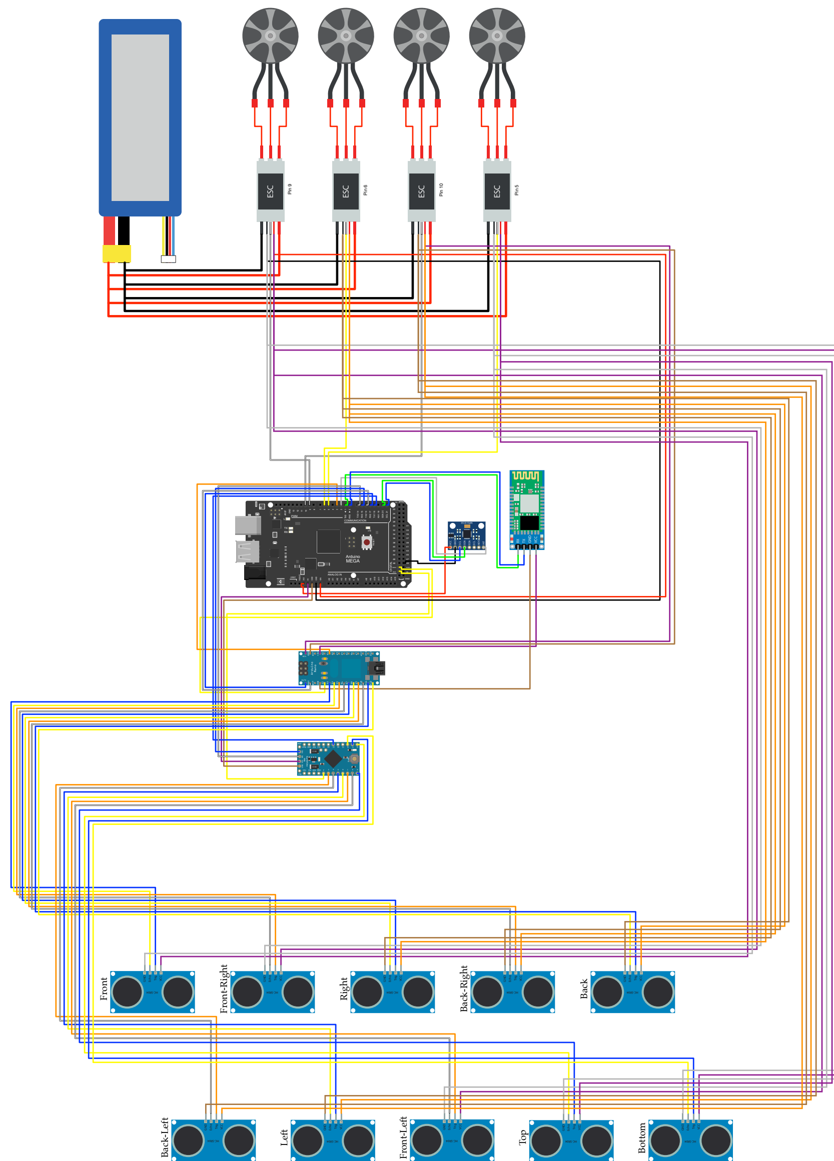


Figure C.1. Wiring of all electric components

Bibliography

[And96] Charles André. SyncCharts: A Visual Representation of Reactive Behaviors. Tech. rep. RR 95–52, rev. RR 96–56. Sophia-Antipolis, France: I3S, Rev. April 1996.

[AOT03] Erdinç Altuğ, James P. Ostrowski, and Camillo J. Taylor. "'Vision Based Control of Model Helicopters'". PhD thesis. Istanbul Technical University, Turkey, Evolution Robotics, USA, and University of Pennsylvania, USA, 2003.

[BBP+07] Samir Bouabdallah, Marcelo Becker, Vincent de Perrot, and Roland Siegwart. "Toward obstacle avoidance on quadrotors". In: 2007.

[BDF+14] Klaus Bengler, Klaus Dietmayer, Berthold Färber, Markus Maurer, Christoph Stiller, and Hermann Winner. "Three decades of driver assistance systems". In: *IEEE Intelligent Transportation Systems Magazine* 6.4 (2014), pp. 6–22.

[Ben13] Paul Benz. "'Implementierung und Evaluierung eines Systems zur Hinderniserkennung und Kollisionsvermeidung für Indoor-Quadrokopter'". PhD thesis. University of Würzburg, Aerospace Information Technology, Apr. 2013.

[GMM12] Nils Gageik, Thilo Müller, and Sergio Montenegro. Obstacle Detection and Collision Acoidance using ultrasonic distance sensors for an autonomous Quadrocopter. Sept. 2012.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "SCCharts: Sequentially Constructive Statecharts for safety-critical applications". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. Edinburgh, UK: ACM, June 2014.

[Höh06] Stephan Höhrmann. "'Entwicklung eines Ultraschall-basierten Ortungssystems für Lego Mindstorms Roboter'". Student research project. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Feb. 2006.

[Lee05] Edward A. Lee. "What are the key challenges in embedded software?" In: *System Design Frontier* 2.1 (2005).

[MHH13] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. "Programming deterministice reactive systems with Synchronous Java (invited paper)". In: *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2013)*. IEEE Proceedings. Paderborn, Germany, 17/18 06 2013.

[MSH14] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. “Compiling SCCharts—A case-study on interactive model-based compilation”. In: *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*. Vol. 8802. LNCS. Corfu, Greece, Oct. 2014, pp. 443–462. DOI: 10.1007/978-3-662-45234-9.

[RAE14] Asher Rahman, Muhammad Farhan Aslam, and Hassan Ejaz. “Gps based navigation and collision avoidance system using ultrasonic sensors and image processing for autonomous vehicle”. In: *International Journal of Computer and Electronics Research* 3.4 (Aug. 2014).

[SMS+15] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Nis Boerge Wechselberg, Carsten Sprung, and Reinhard von Hanxleden. SCCharts: The Railway Project Report. Technical Report 1510. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2015.