Modeling Simulations of Autonomous, Safety-Critical Systems

A Comparison between SCCharts and Ptolemy

Lars Peiler

Bachelor Thesis
2015

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

Advised by
Christian Motika and Steven Smyth

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

Designing an embedded, safety-critical system is a difficult task. Since such a system should never fail under any circumstances, validation is advised. Simulations help to test a system in a safe environment before deploying the system to tests in the real world, therefore following a first-time-right approach that is important for safety-critical systems.

In the scope of a bachelor project, a quadcopter, a small, unmanned aerial vehicle with four rotors, was designed, built and flown. This thesis covers a simulation that was designed to represent the copter and to validate the flight controller. More specifically this means that the quadcopter should be able to fly without crashing by tilting too much in one direction and subsequently flying into a wall.

SCCharts is a visual language designed for specifying safety-critical reactive systems like the quadcopter that uses KIEM to simulate the created model. SCCharts follows a control-flow oriented approach, an approach that focuses on the behavior of a system, yet offers data-flow as well. Data-flow focuses more on the flow of communication and computation between different parts of a model. This thesis will evaluate SCCharts and especially data-flow in SCCharts as a means to design safety-critical systems and to create simulations to execute these systems.

Key words modeling languages, SCCharts, Data-flow, Control-flow, Ptolemy, KIELER, KIEM, simulation, safety-critical system, quadcopter

Contents

List of Figures

Abbreviations

EECS	Electrical Engineering and Computer Sciences
JSON	JavaScript Object Notation
JSSC	Java Simple Serial Connector
KEV	KIELER Environment Visualization
KIELER	Kiel Integrated Environment for Layout Eclipse RichClient
KIEM	KIELER Execution Manager
LGPL	Lesser General Public License
M2M	Model-to-Model
MoC	Model of Computation
RTSYS	Real-Time and Embedded Sytems
SI	Système international d'unités
SCADE	Safety Critical Application Development Environment
SCT	SCCharts Text

Chapter 1

Introduction

Safety-critical systems, such as systems used in the automotive or aerospace industry, are difficult to design. Only one small, faulty part of the system can lead to unforeseen circumstances or crashes and could therefore even cause injuries or worse to humans.

Traditional programming languages and practices generally have problems designing these systems due to unpredictability or the lack of overview. Some scientists like Lee advise against threads as a means to model or create concurrency as non-determinism in programming is fairly dangerous and should be handled explicitly [Lee06]. Therefore, different modeling tools like Ptolemy as well as synchronous languages such as Esterel or Lustre are often used to model and design such systems [BCE+03]. These tools take an approach to eliminate *race conditions* and therefore ensure deterministic behavior. Ptolemy in particular qualifies especially for the simulation of environments due to its heterogeneous approach to modeling. On the other hand, SCCharts, as described below, is a modeling language designed to model reactive systems yet struggles to describe models for environments of bigger physical systems [Uml15]. Recently, SCCharts underwent development in the direction of simulations with the implementation of data-flow. The focus of this thesis now lies on the comparison between Ptolemy and SCCharts as tools for the creation of models on the basis of a model of a quadcopter. The paper will evaluate the possibilities of further development in SCCharts to create a better tool for the creation of simulations and models of environments.

Before going into more detail, the introduction gives a short overview over the most important parts of this thesis: The core modeling technology in SCCharts as well as the simulation tool and the quadcopter itself. Finally, this chapter describes the main problem this thesis focuses on and the general outline of the paper.

1.1 Sequentially Constructive Charts

SCCharts, as introduced by von Hanxleden et al. from the Real-Time and Embedded Sytems (RTSYS) group at the Kiel University [HDM+14], is a visual synchronous language designed for specifying safety-critical reactive systems. SCCharts uses a statechart notation similar to the one used by Harel in his Statecharts approach [Har87]. While SCCharts provides deterministic concurrency based on a synchronous Model of Computation (MoC)

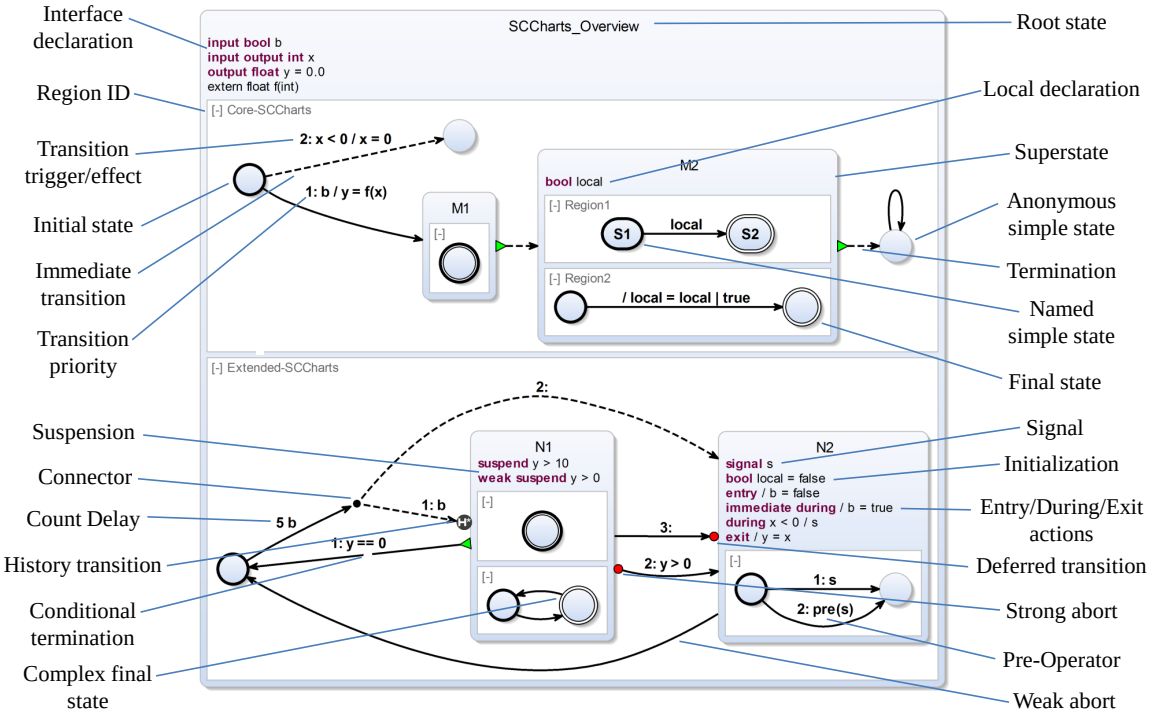


Figure 1.1. Features of Core and Extended SCCharts [HDM+14]

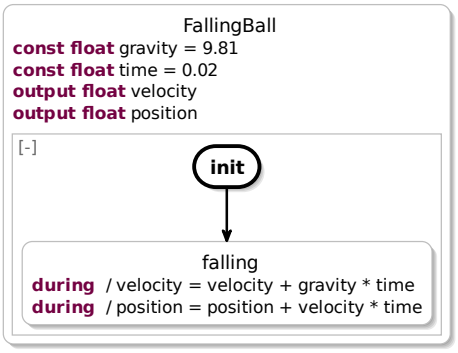
as introduced by Berry [BB91], it also allows emitting different values of a signal (or variable) within one tick as long as the program stays sequentially schedulable. This so called Sequentially Constructive MoC as introduced by von Hanxleden et al. now allows some more programming paradigms while keeping the program deterministic [HMA+13].

SCCharts can be divided into two different parts: Core SCCharts and Extended SCCharts. Core SCCharts contain a minimal amount of elements that is able to express everything the Sequentially Constructive MoC requires, such as states, transitions and hierarchies. Extended SCCharts is built upon Core SCCharts, simplifies a magnitude of elements and adds syntactical sugar. Figure 1.1 shows the different elements of Core and Extended SCCharts. Every additional feature can be translated to equivalent features in Core SCCharts via semantics preserving Model-to-Model (M2M) transformations [HMA+13]. In SCCharts the modeller typically designs the models with the textual language SCCharts Text (SCT). It is a language based on an Xtext grammar, thus allowing content-assist to simplify the

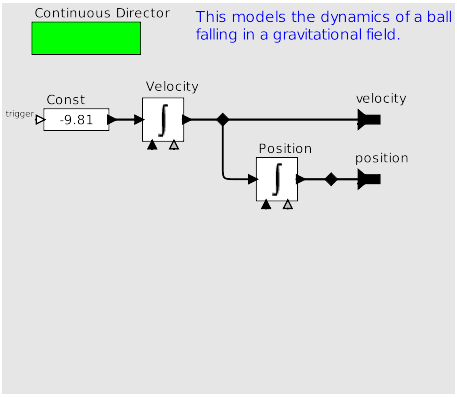
process. Xtext[1] is a framework for development of programming languages and domain specific languages. With SCT the user designs a textual model which is then step for step synthesised into a graphical model as well as compilable C code or other code.

1.1.1 Data-flow in SCCharts

As a derivative of SyncCharts by André [And96], SCCharts was developed with control-flow in mind, using hierarchical, synchronous state machines. Control-flow focuses on descriptions of the behavior of a system. Therefore, control-flow is often visualized using state machines. On the other hand, when trying to compute variables, where the communication between the different components of the model, the so called actors, is a more important aspect, using a data-flow environment might be of advantage [Uml15; CPP05]. For example, to describe a falling ball, a state machine like the one in Figure 1.2a fails to clarify the situation appropriately even in such a small example while the data-flow oriented approach in figure Figure 1.2b is much easier to follow for an outsider.



(a) A falling ball modeled in SCCharts



(b) A falling ball modeled in Ptolemy

Figure 1.2. Difference between a falling ball in a control-flow and a data-flow oriented modeling language

Furthermore, not every model can be completely associated with either data-flow or control-flow. For example, if the same falling ball from above hits the ground, it experiences acceleration upwards. This sudden change in the *state* of the ball can be expressed using different states. In that case, a hybrid system using both concepts is advantageous. Figure 1.3 shows a hybrid system in Ptolemy describing the mentioned situation.

As evident, hybrid systems can express many situations more clearly. Therefore, data-flow in SCCharts was introduced by Umland [Uml15]. It creates a way to use data-flow

[1] http://www.eclipse.org/Xtext/

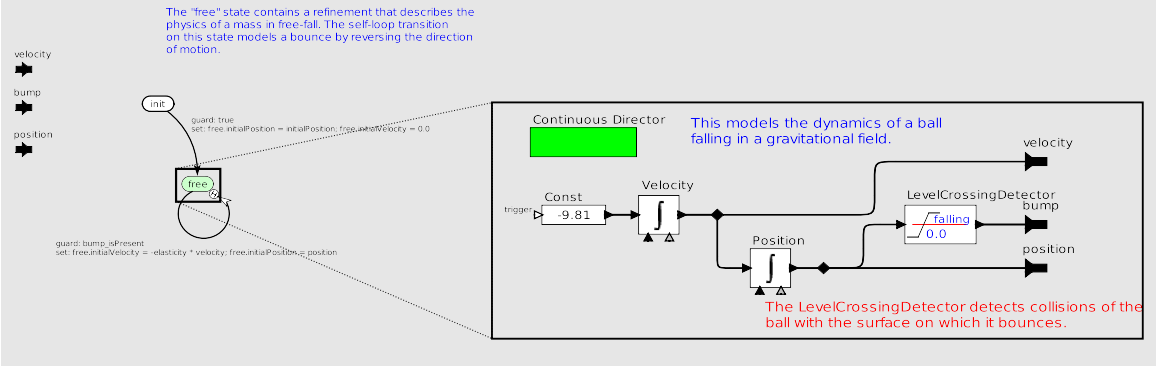


Figure 1.3. A bouncing ball utilizing both control-flow and data-flow in Ptolemy (by E.A.Lee)

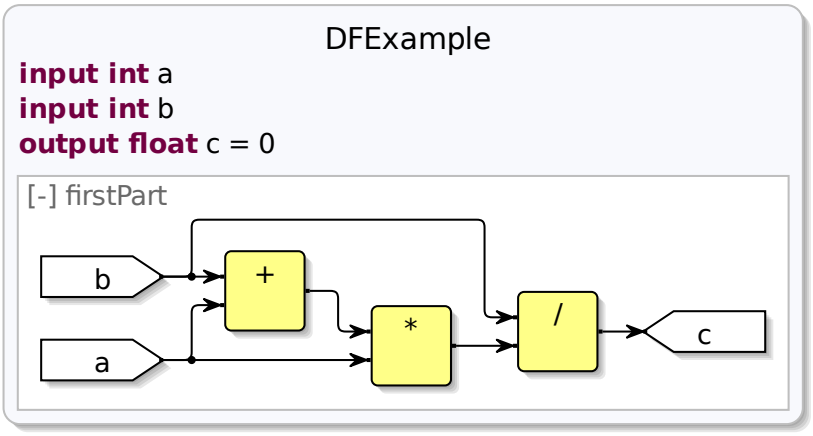


Figure 1.4. An example model of data-flow in SCCharts. The white arrows represent the variables of the system while the yellow boxes visualize the operations carried out on the variables

in the control-flow environment of SCCharts and vice versa. Figure 1.4 shows the chosen visualization in SCCharts. In this approach, everything that is written in a data-flow environment is translated via M2M translations to a core SCChart diagram and can then be used by the SCT compiler to create usable Java code or other code.

1.2 KIELER Execution Manager

SCCharts alone is merely a modeling tool. The modeler then has to use the resulting code from the SCT compiler to create a program running the simulation. This is what the KIELER Execution Manager is responsible for in the KIELER environment. It is an Eclipse infrastructure for managing multiple simulators, visualizators, validators and input/recording/replay facilities at a time introduced by Motika [Mot09]. KIEM calls those

different components of a simulation which then in turn exchange data between each other. This way, instead of creating different programs for each simulator, visualizator, etc. of a project to send data from one part to the other, KIEM operates as a central hub, managing the different parts of the system and their executions.

For each component, the user has to create a *data component*. These data components can be scheduled linearly to create a sequence of executions. Data components *observe* information on a communication bus between these independent data components or *produce* new data which can be used by other components. Every data component also requires a method for the initialization, one for a step and one to wrap the simulation up. Figure 1.5 depicts the schematic layout of a KIEM execution and visualizes the communication bus between the components. These components are modular and can be exchanged at will. Thus, KIEM can be used to test simulations and compare two different simulations of the same object.

In the context of this thesis, KIEM as the simulation tool interfaces data from the simulation to the flight controller of a quadcopter for testing purposes. The modular nature of KIEM allows an easy exchange of the Ptolemy simulation and the SCCharts simulation. Therefore, while evaluating the results, the modeler can compare these results fast and easily. Similarly, KIEM allows an easy exchange of the flight controller. The tests for the controller can be conducted both on the PC running the simulation as well as the quadcopter itself.

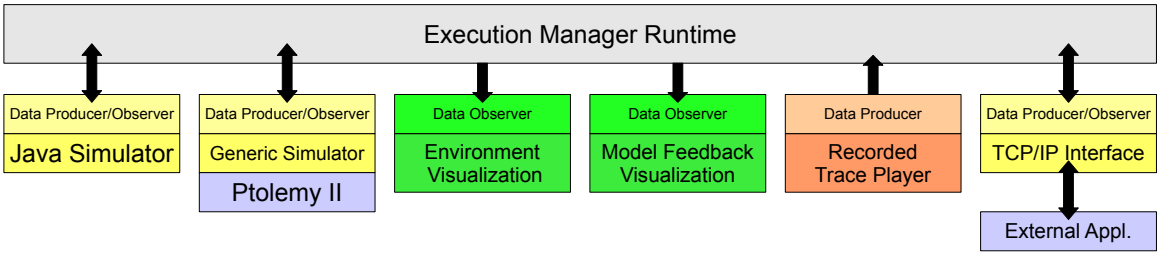


Figure 1.5. Schematic Overview of the KIEM Interface [Mot09]

1.3 The Quadcopter

This thesis was written within the scope of a bachelor project at the RTSYS group with the goal to create a real time, safety-critical system that is modeled as well as tested with the software mentioned above. The created system is a quadcopter, an unmanned aerial vehicle with four rotors, that was supposed to fly autonomously. We divided the project into three distinct parts, one for each student participating in the project: One task was to stabilize the flight of the copter, one part focused on distance measurement and obstacle avoidance while the last part, the simulation, is the matter of this thesis. This way, every

participating bachelor student had a part of the project he was responsible for and was the most prolific in, while everyone could still help working on the other parts. Furthermore, with these responsibilities we ensured that no part of the project was left behind.

Initially we had to learn a lot about aerial vehicles and in particular quadcopters. Before we bought the parts and started building, we gathered information about possible flight controllers, preexisting libraries and projects as well as the parts we needed for our quadcopter. For more information about the building process and the used parts, I recommend the Bachelor theses of Andersen [And15] and Machaczek [Mac15].

At the same time, it was our goal to get the simulation running before the assembly of the copter and the first tests of the flight controller. It was important to us that the testing of the simulation was running on the hardware of the quadcopter itself as this hardware is limited in its computation power. Thus, not only the model itself had to be finished but also the interfacing from and to the quadcopter. Sadly, this was not possible due to time constraints and first tests of the assembled quadcopter were conducted without the help of the simulation. Figure 1.6 depicts the general, final layout of the quadcopter we built. After the initial tests were successful, we started working on the stabilization part, while the creation of the simulation was running concurrently.

For the stabilization we utilized a PID controller. This controller computes the output signals to the motors according to the **P**roportional, the **I**ntegrated and the **D**erivated angles. In the code, different control constants for the P, I and D values for every angle are introduced. They are different for every quadcopter as they depend on the physical properties of the copter like its weight distribution, weight in general and aerodynamic properties. Andersen describes the PID controller and the creation of the flight controller in more detail. This controller was then implemented and tested. During testing, we realized that the copter is slightly unbalanced. This causes that the simulation proposed in this paper does not accurately describe the copter as the simulation assumes a perfectly balanced copter concerning weight distribution.

After we achieved a stable flight, we introduced the distance measurement and obstacle avoidance. For this purpose we used ultrasonic sensors attached to every side and every corner of the copter as well as the top and bottom as Figure 1.6 shows. The obstacle avoidance then uses the flight controller to avoid walls and other objects in the vicinity of the copter. Machaczek [Mac15] describes the implementation of the collision avoidance and distance measurement.

1.4 Problem Description

This quadcopter was vigorously tested. To prevent damage to the quadcopter as well as injuries to bystanders a test can be conducted using a simulation instead of flying the copter in real life. This simulation might not eliminate the chance of a crash, but it can help

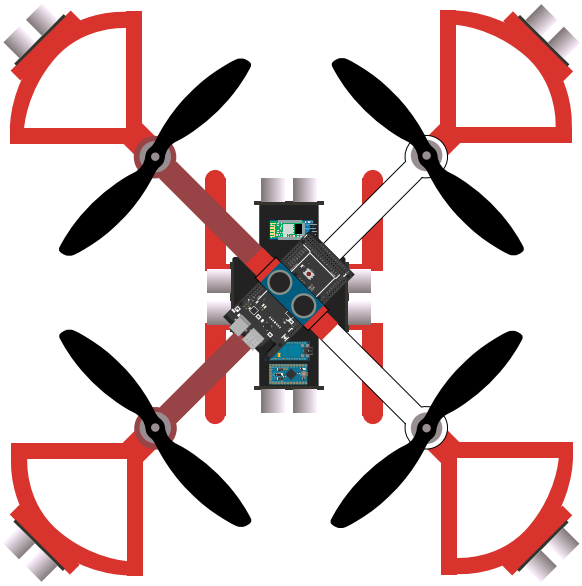


Figure 1.6. Schematic layout of the quadcopter [Mac15]

minimize it. There already exist models of quadcopters and even some simulations, yet they run on software on the PC [Hög14; Luu11]. None of them run embedded on the actual hardware and can therefore not necessarily accurately predict the behavior of the vehicle. This is where not only our simulation but also KIEM is important. An implementation for our simulation using KIEM creates an easy way to interface from and to the copter and helps to compare and evaluate different approaches via the modularity of KIEM as well.

This furthermore helps us in comparing SCCharts with Ptolemy. Since it is easy to exchange the SCCharts data component with the Ptolemy data component, it is also easy to compare the two models in the simulation. Still this is not sufficient concerning the evaluation. With data-flow in SCCharts being a new addition, it has not been tested and evaluated much. The goal of this thesis is to compare it to Ptolemy, which is already an established tool for the creation of models and simulations, and look for possibilities to enhance SCCharts as a tool for models of simulations.

1.5 Outline

This thesis begins with a short overview over related work in Chapter 2. Chapter 3 introduces important technologies used in the making of the model, the simulation, and the evaluation. It will contain information about Ptolemy, the tool SCCharts will be compared with, as well as information about the hardware and software used in the process of this thesis.

In Chapter 4, the physical model will be described in detail. The chapter will give insight into the different aspects of the model such as the rotational matrices, how to calculate the acceleration, velocity and position as well as the angular velocities. Additionally, the reasons for the use of KIEM as the simulation tool will be explained in more detail. Afterwards, in Chapter 5, the realization of the model in both Ptolemy and in SCCharts is delineated. Both of these sections will cover the same model realized on different platforms. Furthermore, this chapter will deal with the implementation of the simulation and how the models have been tested. A short section about calculations for some physical properties of the quadcopter will close the chapter.

Chapter 6 will evaluate the two approaches and will compare the advantages and disadvantages of those. It will give insight into the creation of the models in the different tools and the problems encountered. The chapter will further describe the differences in the calculation, as the two tools are inherently different in their computations.

The final chapter covers the conclusion, lessons learned in Ptolemy and SCCharts as well as some suggestions for future work in SCCharts, especially data-flow in SCCharts.

Chapter 2

Related Work

This chapter covers other scientific papers and related work. First, it addresses papers that are also about the simulation and modeling of a quadcopter. Following that, the chapter describes different software that could be considered for the creation of a model or a simulation for a quadcopter or a similar vehicle.

2.1 Other Models and Simulations

There have already been multiple papers and theses describing physical properties and models of quadcopters of different sizes. These physical properties are fairly well understood and are described in a multitude of papers.

Höger [Hög14] describes the model of quadcopters in general and then adapts this model to the *Crazyflie*[1], a very small quadcopter for indoor use. His work is more condensed concerning the model and focuses more on the mathematical backgrounds of the calculations and the adaptation required for the Crazyflie. Thus, he concentrates a lot on determining unknown constants of the quadcopter. His approach to finding these values is very complex and doing a similar approach would go beyond the scope of this thesis. Therefore, only the model and information about the rotational matrices were taken from this thesis. Höger uses Matlab as the modeling and simulation environment. Thusly, the calculations do not run in the native code the hardware of the copter requires. This could possibly lead to unexpected behavior if the code is finally running on the copter as it is not tested, because of the speed of the hardware or because of differences in the native language.

Another approach to model a generic quadcopter has been done by Luukkonen [Luu11]. He describes the physical model of the quadcopter in more detail and furthermore gives insight into his method for controlling the copter. His approach was never tested in reality. Furthermore, he uses values to input into his simulation that a common quadcopter cannot accurately measure such as the current velocity as an input to the copter. The copter might be able to calculate his velocity with the help of the accelerometer, but common accelerometers are not very exact and often output noise. He also uses Matlab as the modeling and simulation environment.

[1] https://www.bitcraze.io/crazyflie-2/

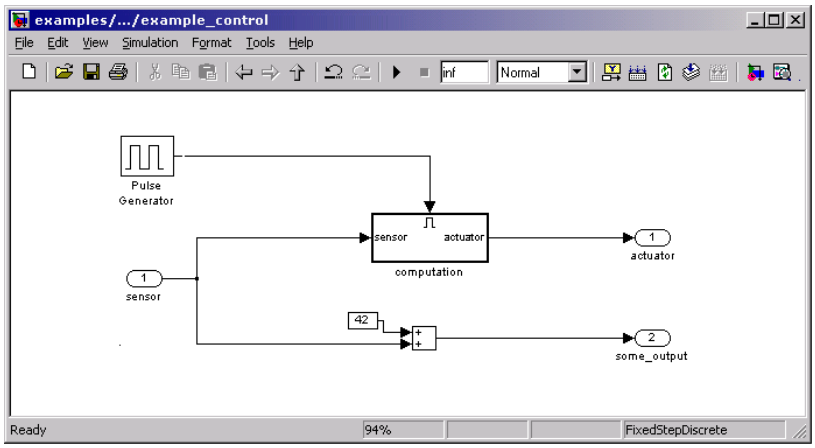


Figure 2.1. The simulation GUI of Matlab/Simulink [Mot07]

McGilvray and Tayebi describe the physical properties of a quadcopter in their paper *Attitude stabilization of a four-rotor aerial robot* [TM04]. Furthermore, they show a small part of their simulation results using their model and the corresponding PID controller. Since many physical properties of a quadcopter are hard to determine, the simulation in this thesis uses the properties of the quadcopter described by McGilvray. Therefore it is hard to apply the simulation results to the quadcopter that was built for the project of this thesis.

2.2 Simulation and Modeling Tools

Since one big part of this thesis is the evaluation of SCCharts as a modeling environment and simulation tool for data-flow oriented models, it is also important to consider different comparison tools. For this purpose, Ptolemy, as described in more detail in Chapter 3, was chosen due to the preexisting integration in KIEM and the ease to create heterogeneous models, which simulations often times require. However, other tools should still be considered. This section covers some popular tools that are used to design and model simulations.

2.2.1 Matlab and Simulink

Both Höger and Luukkonen use Matlab[2] as a tool for both modeling and simulating their approach. Matlab is a programming language designed specifically with mathematical systems in mind. This includes data-flow oriented systems like the physical model of a quadcopter. Matlab also allows for very clear visualizations of simulation results. As such, it is very qualified for the purpose of this paper. However, Matlab lacks when it comes

[2] http://www.mathworks.com/products/matlab/

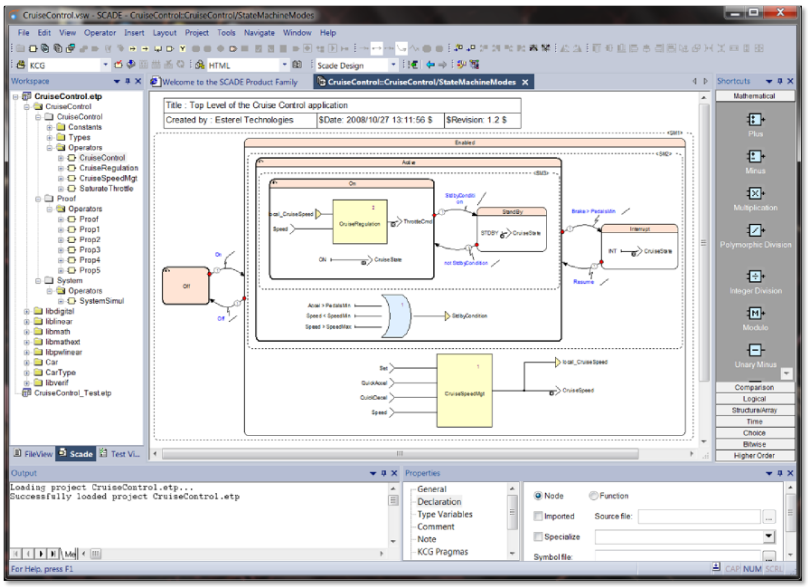


Figure 2.2. The GUI and an example project in the SCADE Suite[5]

to usability and is definitely neither beginner friendly nor clear when modeling bigger systems. Therefore, Matlab alone was not considered. Simulink[3], as depicted in Figure 2.1, extends Matlab with a visualization and a graphical editing tool for data-flow models. It simplifies a lot of the problems Matlab alone struggles with mentioned above. However, the integration of Matlab and Simulink into KIEM is difficult as a stepwise execution of models is only possible after computing complete results. This means that a real-time execution of the simulation is not possible in Simulink. Therefore, it is impossible for Matlab and Simulink to react to the behavior of the flight controller, if the flight controller is not written in Matlab as well. Since we were already experienced with SCCharts as well as Ptolemy, we chose to disregard Matlab. Another reason against Matlab and Simulink and for SCCharts and Ptolemy is that SCCharts and Ptolemy are open source while Matlab is a commercial product that is not open source.

2.2.2 Safety Critical Application Development Environment

The SCADE Suite[4], as depicted in Figure 2.2, is a model-based development environment for critical embedded software. Modelers can design their model both graphically as well as textually with SCADE, which can then generate C or ADA code from these models. Motika [Mot07] uses this tool to design a simulation for a model railway that is supposed to

[3] http://www.mathworks.com/products/simulink/

[5] Source: http://www.esterel-technologies.com/wp-content/uploads/2015/03/SCADE-Suite-IDE-View.png

[4] http://www.esterel-technologies.com/products/scade-suite/

control multiple trains on different tracks and courses. Since KIEM was chosen to simulate the model and SCADE interfaces using the Transmission Control Protocol (TCP) which complicates the communication, SCADE was not considered as an alternative to Ptolemy. Since SCADE is also not open source, another way to integrate the tool into a work-flow is not possible.

2.2.3 Testing in a Safe Environment

Multiple studies with quadcopters were conducted in a safe environment instead of using a simulation. One such approach is described by Castillo et al. [CLD05]. However, the required hardware for the testing environment was not available. Furthermore, in the tests conducted in the scope of this thesis, it was noticed, that a cable attached to the copter influences his flying capabilities severely.

The project still used a lot of live testing without this testing environment to test the actual capabilities of the quadcopter. These tests were conducted without an environment like the one mentioned above and will not be a part of this thesis.

Chapter 3

Used Technology

Prior to explaining the model of the quadcopter, it is necessary to sketch the important used technologies in the making of this thesis. This chapter contains information about a tool used to create a model that compares to the one created in SCCharts, more general information about the KIELER project as well as technology used for the exchange of information to and from the quadcopter and the hardware on the quadcopter.

3.1 Ptolemy

The Ptolemy Project[1] is an open-source, Java based tool for designing and simulating concurrent, real-time, embedded systems as introduced by [Lee03]. Ptolemy II, hereafter just Ptolemy, is in development by the Electrical Engineering and Computer Sciences (EECS) faculty at UC Berkeley since 1996. This tool simplifies the creation of complex models with hierarchical structures and an actor-oriented design. Actors are concurrently executing software components that, when interconnected via ports, send messages to other actors and thus make up a model. The behavior of every actor is described in its Java class and ranges from simple mathematical operations like adding and subtracting to complex ones like integration.

A Ptolemy model is stored in an XML-file. Thus, a user can easily extract information if he wants to. The semantics of a model is determined by its director – a component implementing a model of computation. Each level in the hierarchy of the model can have its own director, and understanding these directors and their interactions is a core part in the creation of different models with Ptolemy. These different directors allow different approaches to modeling such as a data-flow oriented or a control-flow oriented approach. Heterogeneous combinations of directors enable the modeler to design the model he or she wants to create. An example for this heterogeneous approach is the use of a synchronous reactive director, a director using a data-flow oriented approach, in combination with a modal model, a control-flow oriented director, that is hierarchically embedded in the model with the synchronous reactive director, creating a StateChart [Pto14].

Vergil, the UI of Ptolemy, visualizes the model as seen in Figure 3.1 and allows the modeler to easily edit the model. The user controls the tool by drag-and-drop and can edit

[1] http://ptolemy.eecs.berkeley.edu/

actors and composite models as well as open them. While a composite model is then again opened in another Vergil window and can be edited, opening an actor displays the java class of the selected actor describing the behavior of it.

The layout of a model in Ptolemy can be automatically generated by a layout algorithm or created by the modeler himself.

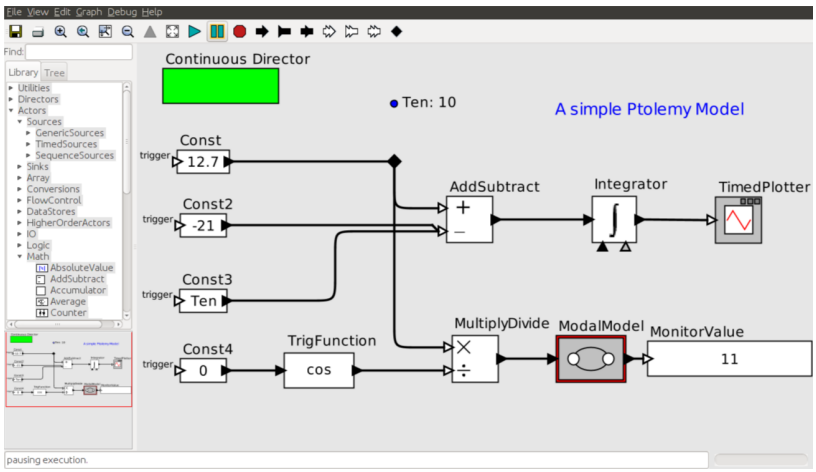


Figure 3.1. The Ptolemy GUI, using Vergil

3.1.1 KielerIO

Furthermore, to input data to Ptolemy, for example from KIEM, the simulator needs an actor that is not implemented in Ptolemy, the *KielerIO*. This actor allows both regular use as a constant source and as a source that inputs data coming from KIEM or other similar simulators. As KIEM is already simulating the SCCharts model, it was an easy decision to use the already preexisting infrastructure and keep everything in the same place. Using KIEM also ensures that there are no differences between the simulation using Ptolemy and using SCCharts. This actor has been introduced in detail by Motika in his diploma thesis [Mot09]. For the purpose of this thesis, the KielerIOFloat actor was created. It has the same functionality as the KielerIO but is able to input floats and doubles instead of integers.

3.2 Kiel Integrated Environment for Layout Eclipse RichClient

The KIELER project [2] is an academic research project from the Real Time and Embedded Systems group at the Kiel University. Its aim is to enhance the graphical, model-based design

[2] http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Overview

of complex systems. It is an open-source project under the Eclipse Public License (EPL) and is separated into four different areas: Semantics, Pragmatics, Layout and Demonstrators.

The semantics department of the project deals with execution semantics for meta models. It covers compiling as well as simulating models. The already mentioned SCCharts and KIEM projects are part of the semantics department.

Pragmatics and layout cover the practical aspects of a modelers work. Enhancing the creation and modification process of models for a modeler is the core of these areas. Thus, a big part of these branches is automatic layout design of diagrams and graphs as well as efficient layout algorithms.

Lastly, the demonstrators are editors used for testing and to demonstrate developments from the other branches.

Figure 3.2. Overview over the KIELER project

3.3 Java Simple Serial Connector

The Java Simple Serial Connector (JSSC) v.2.8.0[3] is an open-source project published under the Lesser General Public License (LGPL). It is a library for working with serial ports in Java and supports every current major OS. In this project it is required to communicate between the simulator and the flight controller. Since we wanted to test the software while it is running on the controller, we needed a way to input and output data between the simulation running on a PC and the flight controller.

Compared to other libraries for serial communication in Java, JSSC provides many simplifications of procedures, for example opening a port. Furthermore, other projects like RxTx or Oracles own JavaComm are currently not in development anymore. Another

[3] https://code.google.com/p/java-simple-serial-connector/

reason for JSSC is that the current version of RxTx does not support Java 8 – at least without further modifications. JavaComm on the other hand is pretty much unusable and should not be considered for serial communication in Java. It has not been properly updated in ten years and official support has stopped. Therefore, even finding an official download link proves to be difficult.

After completing the program, I have heard of a relatively new java serial library called jSerialComm[4]. It is a platform independent library for serial communication with java. It claims to be very lightweight and efficient. A closer look into this library might be advised for future use.

3.4 Arduino Mega 2560

JSSC then communicates with software running on the quadcopter, more precisely on the Arduino Mega 2560 board, for simplification in the following called Arduino, on the quadcopter. This flight controller is running on custom libraries and SCCharts generated code as the flight controller. The Mega 2560 is a microcontroller board based on the ATmega2560 [5] with 54 digital input/output pins, 16 analog inputs, four hardware serial ports, a USB connection and more. The board can be supplied with power via the USB serial port as well as via battery over an input pin. Using the Arduino Software IDE [6], programs in C++ or the Arduino own Arduino language can be uploaded to the board.

We have chosen this board as we deemed it powerful enough while not being too heavy, requiring too much voltage to run or not being able to output current at the desired voltage for our sensors. It also provides enough pins for our purposes. Too small boards might not have enough pins or might not run fast enough to calculate the flight properties in time. Furthermore, there was already a spare Arduino Mega in the office so the decision was easy to make.

Another alternative to an Arduino board was using a Raspberry Pi[7] or a BeagleBone[8] computer. These boards, as opposed to the Arduino boards, require an operating system to run. Thus, they are not necessarily suitable as safety-critical systems as a safety-critical system has to react as fast as possible to the environment and an operating system might interrupt critical computations with OS specific functionalities. This can lead to unpredictable behavior.

Andersen [And15] as well as Machaczek [Mac15] contain more information about the flight controller and the hard- and software of the quadcopter.

[4] http://fazecast.github.io/jSerialComm/

[5] http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf

[6] https://www.arduino.cc/en/Main/Software

[7] https://www.raspberrypi.org/

[8] http://beagleboard.org/

Chapter 4

Model and Simulation

To be able to compare Ptolemy and SCCharts, a model to simulate is required. Therefore, this chapter explains the physical and mathematical properties of the quadcopter. Section 4.1 contains all the equations and their backgrounds, explaining the model in detail. It also explains the rotations required to describe the model accurately and afterwards the properties pertaining to the acceleration as well as to the angles. The section concludes with information about the physical properties of the simulated quadcopter. Following that, Section 4.2 discusses how to integrate the simulation into the setup of the simulation using KIEM.

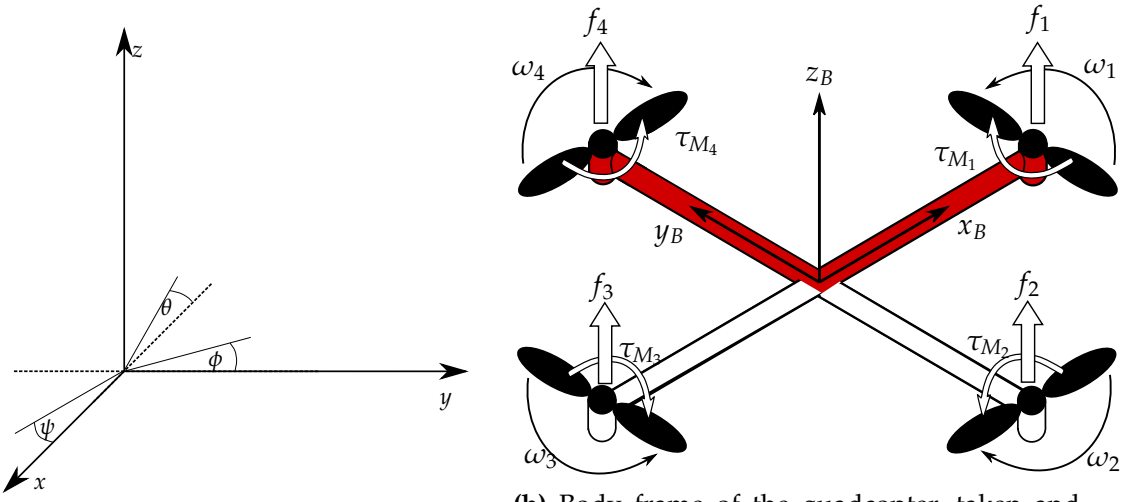
4.1 Mathematical Model

Before going into detail, this section explains a few necessities for the model. As shown in Figure 4.1, the quadcopter has to be viewed in two different frames: First, the inertial frame, seen on the left, is the frame of the room. It is fixed and cannot change. Second, the body frame, as seen on the right, is the frame of the copter. Its axes, the body axes x_B, y_B and z_B, are fixed to the arms and body of the quadcopter and its point of origin is the center of mass, so it moves with the copter at all times. These frames of reference are necessary, as the simulation will send data calculated from both frames to the Arduino, which Section 4.2 describes in more detail. The inertial frame is important for the location of the vehicle and thus for the distances to the walls and the body frame is needed for gyroscopic values like the angular velocities or the linear acceleration of the quadcopter. Furthermore, to compute these values, calculations in both frames are required as will be evident from the descriptions in this chapter.

In the sections pertaining to the angles and angular velocities, Newton-Euler equations [Hah02] describe the behavior of the copter. Since the copter is assumed to be a rigid body, these equations can be used to describe the dynamics of the quadcopter [Luu11].

4.1.1 Fundamental of the model

The absolute linear position of the quadcopter is defined in the inertial frame of the quadcopter by $\xi = \begin{pmatrix} x & y & z \end{pmatrix}^T \in \mathbb{R}^3$ and the Euler angles describing the rotation of the quadcopter are defined by $\eta = \begin{pmatrix} \phi & \theta & \psi \end{pmatrix}^T \in \mathbb{R}^3$ with the roll angle ϕ, the pitch



(a) Inertial frame of the quadcopter

(b) Body frame of the quadcopter, taken and adapted from Höger [Hög14]

Figure 4.1. The different frames of the quadcopter

angle θ and the yaw angle ψ. ϕ turns around the x-axis, θ around the y-axis and ψ around the z-axis as can be seen in Figure 4.1a. These three angles are important for the conversion of values from one frame to the other.

There are multiple conventions for the Euler angles [Gol80]. This paper will use the zyx convention commonly used when studying the properties of vehicles, especially aerial vehicles [Hög14].

In the body frame, the angular velocities ν are given by $\nu = \begin{pmatrix} p & q & r \end{pmatrix}^T \in \mathbb{R}^3$ with p being the angular velocity around the x_B-axis, q around the y_B-axis and r around the z_B-axis of the body frame. These angular velocities are positive if the copter is turning clockwise around the axis while facing the direction of the axis.

4.1.2 Rotation

Since there are two different frames of reference this section will explain a way to convert a vector in one frame to a vector in another frame. For this purpose, rotational matrices are applied to these vectors. Since the model uses the zyx-convention, a vector has to be rotated in a specific way in order to convert it from the body frame to the inertial frame: First, the vector has to rotate around the z-axis by ψ, then it has to rotate around around the y-axis by θ and finally around the x-axis by ϕ.

Applying these rotations

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{pmatrix}$$

$$R_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{pmatrix}$$

and

$$R_z(\psi) = \begin{pmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

to a vector l and its representative in the body frame l_B in the correct order leads to the following equations:

$$\begin{aligned} l &= R_x(\phi)R_y(\theta)R_z(\psi)l_B \\ &= \begin{pmatrix} C_\theta C_\psi & C_\theta S_\psi & -S_\theta \\ S_\phi S_\theta C_\psi - C_\phi S_\psi & S_\phi S_\theta S_\psi + C_\psi C_\phi & S_\phi C_\theta \\ C_\phi S_\theta C_\psi + S_\phi S_\psi & C_\phi S_\theta S_\psi + S_\phi C_\psi & C_\phi C_\theta \end{pmatrix} l_B \\ &= R_\eta l_B \end{aligned} \tag{4.1.1}$$

with $S_\alpha = \sin(\alpha)$ and $C_\alpha = \cos(\alpha)$.

The rotational matrix R_η is orthogonal, thus $R_\eta^{-1} = R_\eta^T$. This is the rotational matrix from the inertial frame to the body frame.

These two rotational matrices cannot translate the angular velocities from one frame to the other, as these rotations behave differently. For example, the angular velocity around the z-axis never changes, no matter how the copter is turned. Thus, the yaw angle undergoes two rotations, the pitch angle one rotation and the roll angle no rotations. The resulting matrix W_η translates changes in the Euler angles η with $\theta \neq \frac{\pi}{2}$ to angular velocities and the inverse matrix W_η^{-1} maps the angular velocities to changes in the Euler angles:

$$\begin{aligned} \nu &= \begin{pmatrix} p \\ q \\ r \end{pmatrix} = R_3(\phi)R_2(\theta) \begin{pmatrix} 0 \\ 0 \\ \dot{\psi} \end{pmatrix} + R_3(\phi) \begin{pmatrix} 0 \\ \dot{\theta} \\ 0 \end{pmatrix} + \begin{pmatrix} \dot{\phi} \\ 0 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & -\sin(\theta) \\ 0 & \cos(\phi) & \sin(\phi)\cos(\theta) \\ 0 & -\sin(\phi) & \cos(\phi)\cos(\theta) \end{pmatrix} \begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} \\ &= W_\eta \dot{\eta} \end{aligned} \tag{4.1.2}$$

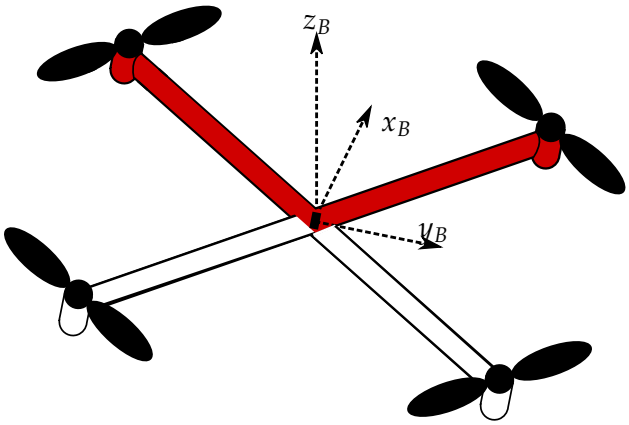


Figure 4.2. The axes of the gyroscope in the body frame depending on the position of the gyroscope with the x_{Body}, y_{Body} and z_{Body} axes (short: x_B, y_B, z_B)

$$\begin{aligned} \dot{\eta} &= W_\eta^{-1}\nu \\ &= \begin{pmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \frac{\sin(\phi)}{\cos(\theta)} & \frac{\cos(\phi)}{\cos(\theta)} \end{pmatrix} \begin{pmatrix} p \\ q \\ r \end{pmatrix} \end{aligned} \tag{4.1.3}$$

Importantly, $\theta \neq \frac{\pi}{2}$ is necessary, both because $\cos(\frac{\pi}{2}) = 0$, and $\tan(\pi/2)$ is undefined. Realistically though, this state of the quadcopter should never be achieved.

Lastly, as the gyroscope has been mounted on the quadcopter in a way such that the angles do not coincide with the angles from the model as depicted in Figure 4.2, it will be necessary to rotate a vector l by 45°. To accomplish that, the model uses a rotational matrix once more, this time only rotating around the z_B-axis:

$$l_{rot} = Y_{45^\circ} l = \begin{pmatrix} \cos(\frac{\pi}{4}) & -\sin(\frac{\pi}{4}) & 0 \\ \sin(\frac{\pi}{4}) & \cos(\frac{\pi}{4}) & 0 \\ 0 & 0 & 1 \end{pmatrix} l \tag{4.1.4}$$

4.1.3 Linear Accelerations, Velocities and Position

The gyroscope in the quadcopter provides values for the linear acceleration. Therefore, the simulation has to output this value as well. The linear acceleration is a vector that is determined via the thrust T of the vehicle in the direction of the z_B-axis. Every rotor exerts a force f_i in the direction of its rotor z-axis as can be seen in Figure 4.1b. Since the body of the copter is rigid, all four rotors exert a force in the same direction. The totaled force of all four rotors is the thrust of the copter. This is calculated by

$$f_i = l\omega_i^2 \tag{4.1.5}$$

$$T = \sum_{i=1}^{4} f_i = l \sum_{i=1}^{4} \omega_i^2 \tag{4.1.6}$$

with l being the *lift constant*[1] and ω_i being the angular velocity of the rotor corresponding to the motor i. The calculated thrust exerts constantly along the z-axis of the quadcopter and is therefore in the body frame. By applying the rotational matrix R_η from Equation 4.1.1 to the vectorized thrust, we can translate this force into the inertial frame. Additionally, gravity exerts on the quadcopter. Therefore it has to be subtracted from the acceleration provided by the thrust.

$$\begin{aligned} a &= \begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -g \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \frac{T}{m} \end{pmatrix} R_\eta \\ &= \begin{pmatrix} 0 \\ 0 \\ -g \end{pmatrix} + \frac{T}{m} \begin{pmatrix} \cos(\phi)\sin(\theta)\cos(\psi) + \sin(\phi)\sin(\psi) \\ \cos(\phi)\sin(\theta)\sin(\psi) - \sin(\phi)\cos(\psi) \\ \cos(\phi)\cos(\theta) \end{pmatrix} \end{aligned} \tag{4.1.7}$$

This acceleration though does not yet factor in aerodynamic effects. Thus, the following equation describes these effects with the drag force coefficients A_x, A_y and A_z and the velocity v slowing the acceleration down.

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = a - \frac{1}{m} \begin{pmatrix} A_x & 0 & 0 \\ 0 & A_y & 0 \\ 0 & 0 & A_z \end{pmatrix} v \tag{4.1.8}$$

The velocity v has to be known to calculate the acceleration, which in turn has to be known to calculate the velocity. Thus, a loop-back is required.

Now that the linear acceleration is known, this value is integrated over time to calculate the velocity v and integrated once more to calculate the current position ξ of the quadcopter.

4.1.4 Angular Velocities and Angles

To actually apply the rotational matrices mentioned above, the Euler angles η have to be computed. Thus, it is necessary to calculate the torques τ_ϕ, τ_θ and τ_ψ in the direction of the corresponding body frame axes:

[1] More about the lift and drag constants can be read here:
http://mragheb.com/NPRE475WindPowerSystems/AeorodynamicsofRotorBlades.pdf

$$\tau = \begin{pmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{pmatrix} = \begin{pmatrix} d_r l(-\omega_2^2 + \omega_4^2) \\ d_r l(-\omega_1^2 + \omega_3^2) \\ \sum_{i=1}^{4} \tau_{M_i} \end{pmatrix} \tag{4.1.9}$$

with d_r being the distance between the rotor middle and the center of mass of the quadcopter, l being the *lift constant*[1] mentioned before and

$$\tau_{M_i} = b\omega_i^2 + I_M \dot{\omega}_i \tag{4.1.10}$$

where b is the *drag constant*[1] of the rotors and I_M is the inertia moment of the rotors. Since the rotor acceleration $\dot{\omega}_i$ as well as the inertia moment I_M is very small, it is omitted.

This implies that movement in the roll direction can be increased by increasing the velocity of the fourth rotor and/or decreasing the velocity of the second rotor. Likewise, pitch direction movement can be increased by increasing the velocity of the third rotor and/or decreasing the velocity of the first rotor. Turning around the z_B-axis can be achieved by increasing opposite rotors and decreasing the other two rotors. These movements can obviously be combined to create a smooth turning of the copter if wanted.

In the body frame, the angular acceleration $I\dot{\nu}$ together with the centripetal forces $\nu \times (I\nu)$ and the gyroscopic forces Γ equal the torque τ. Thus, with a little bit of reorganizing, the following equation can be formulated:

$$I\dot{\nu} + \nu \times (I\nu) + \Gamma = \tau \tag{4.1.11}$$

$$\begin{aligned} \dot{\nu} &= I^{-1} \left(- \begin{pmatrix} p \\ q \\ r \end{pmatrix} \times \begin{pmatrix} I_{xx} p \\ I_{yy} q \\ I_{zz} r \end{pmatrix} - I_r \begin{pmatrix} p \\ q \\ r \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \omega_\tau + \tau \right) \\ &= \begin{pmatrix} \frac{1}{I_{xx}} \\ \frac{1}{I_{yy}} \\ \frac{1}{I_{zz}} \end{pmatrix} \left(\begin{pmatrix} (I_{yy} - I_{zz}) qr \\ (I_{zz} - I_{xx}) pr \\ (I_{xx} - I_{yy}) pq \end{pmatrix} - I_r \omega_\tau \begin{pmatrix} q \\ -p \\ 0 \end{pmatrix} + \begin{pmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{pmatrix} \right) \end{aligned} \tag{4.1.12}$$

with $\omega_\tau = \omega_1 - \omega_2 + \omega_3 - \omega_4$.

The derivative of the angular accelerations $\dot{\nu}$ from the equation above can now calculate the angular velocities ν in the body frame by integrating them over time. These values are an output value of the gyroscope.

Furthermore, the angular velocities ν, translated to the inertial frame with the help of the rotation matrix W_η^{-1} from Equation 4.1.2, when integrated, result in the current angles η.

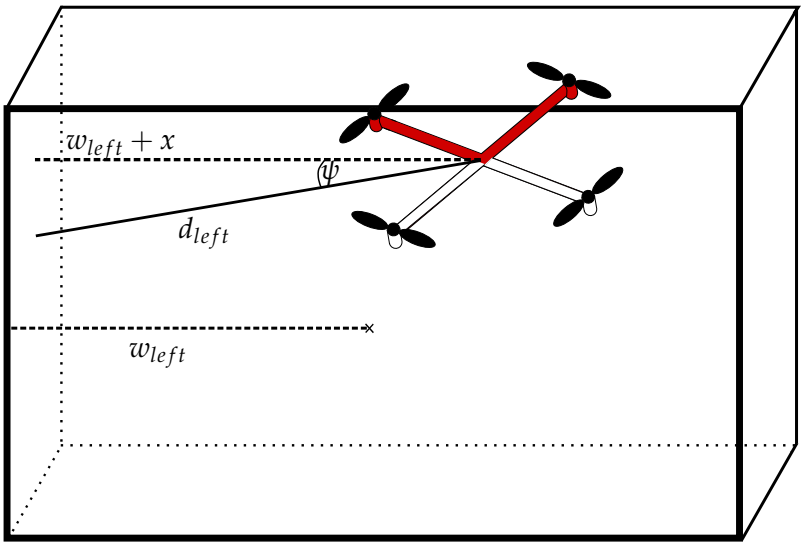


Figure 4.3. Distance measurement of a quadcopter at the position (*x*,*y*,*z*)

$$\begin{aligned} \dot{\eta} &= \begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} = W_\eta^{-1} \nu \\ &= \begin{pmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \frac{\sin(\phi)}{\cos(\theta)} & \frac{\cos(\phi)}{\cos(\theta)} \end{pmatrix} \begin{pmatrix} p \\ q \\ r \end{pmatrix} \end{aligned} \tag{4.1.13}$$

The Euler angles η allow the accurate use of the rotational matrices.

4.1.5 Distance Calculation

To calculate the distances $d = \begin{pmatrix} d_{front} & d_{right} & d_{back} & d_{left} & d_{up} & d_{down} \end{pmatrix}^T \in \mathbb{R}^6$ between the copter and the walls, the model requires information about the room. This is provided by the distance between the walls $w = \begin{pmatrix} w_{front} & w_{right} & w_{back} & w_{left} \end{pmatrix}^T \in \mathbb{R}^4$ and the center of the room as well as the height w_{roof}. The walls of the room are assumed to be rectangular with the quadcopter initially being in the center of the room. At the start of the simulation the arms of the quadcopter point to the corners of the room. Figure 4.3 depicts the general layout of the room after the quadcopter has been flying for a short time.

The calculations for the distances use a few simplifications minimizing the stress on the simulations, since especially Ptolemy started to slow down during the calculations. First, the simulation assumes that the roll and pitch angles will remain in a fairly stable range close to 0°. This means that the quadcopter remains in a stable flight position. Therefore,

Table 4.1. The different distance calculations for every state of the copter in every direction

Distance sensor	Facing front $-\frac{\pi}{4}$ to $\frac{\pi}{4}$	Facing right $\frac{\pi}{4}$ to $3\frac{\pi}{4}$	Facing back $3\frac{\pi}{4}$ to $-3\frac{\pi}{4}$	Facing left $-3\frac{\pi}{4}$ to $-\frac{\pi}{4}$
d_{front}	$\frac{w_{front}-y}{\cos(\psi)}$	$\frac{w_{right}-x}{\cos(\psi-\frac{\pi}{2})}$	$\frac{w_{back}+y}{\cos(\psi-\pi)}$	$\frac{w_{left}+x}{\cos(\psi+\frac{\pi}{2})}$
d_{right}	$\frac{w_{right}-x}{\cos(\psi)}$	$\frac{w_{back}+y}{\cos(\psi-\frac{\pi}{2})}$	$\frac{w_{left}+x}{\cos(\psi-\pi)}$	$\frac{w_{front}-y}{\cos(\psi+\frac{\pi}{2})}$
d_{back}	$\frac{w_{back}+y}{\cos(\psi)}$	$\frac{w_{left}+x}{\cos(\psi-\frac{\pi}{2})}$	$\frac{w_{front}-y}{\cos(\psi-\pi)}$	$\frac{w_{right}-x}{\cos(\psi+\frac{\pi}{2})}$
d_{left}	$\frac{w_{left}+x}{\cos(\psi)}$	$\frac{w_{front}-y}{\cos(\psi-\frac{\pi}{2})}$	$\frac{w_{right}-x}{\cos(\psi-\pi)}$	$\frac{w_{back}+y}{\cos(\psi+\frac{\pi}{2})}$
d_{up}	$w_{roof}-z$	$w_{roof}-z$	$w_{roof}-z$	$w_{roof}-z$
d_{down}	z	z	z	z

only the yaw angle ψ and the x and y coordinates of the current position ξ is necessary to calculate the distance to the walls and only the z coordinate is required to calculate the distance to the floor and the roof. Second, as Figure 4.4 illustrates, it is hard to measure the distance to the actual closest wall as every sensor would have to calculate his distance to every of the four walls and then comparing all distances creating a lot of redundant information and calculations. To counteract this, the simulation assumes that, when a sensor is facing the general direction of a wall, it only calculates the distance to this wall. As can be seen in Figure 4.4 this does not necessarily compute the distance to the actual closest wall and therefore the distance a real distance sensor would detect. Yet, another sensor would detect this wall with a closer distance then the first sensor, signalizing the copter to back off from this wall. Therefore, these calculations should be sufficient.

With these simplifications multiple different equations were developed for each of the four directions the copter can face. Table 4.1 shows these equations.

Another reason for this heuristic was to create a possibility to use both data-flow and control-flow in the model. This approach makes it possible for the model to compare the implementation of heterogeneous models in Ptolemy and SCCharts.

4.1.6 Adjusting Output Values to the Quadcopter

There are multiple different peculiarities to consider when transmitting the values to the Arduino. First, the acceleration calculated in Section 4.1.3 are values in the inertial frame. Thus, they have to be converted to the body frame with the inverse of the rotational matrix from Equation 4.1.1. Furthermore, since gravity constantly exerts a force on the gyroscope, it measures an acceleration of $-g$ in the direction of the ground at all times, even if the copter is completely stationary. As such, before the conversion to the body frame, we need

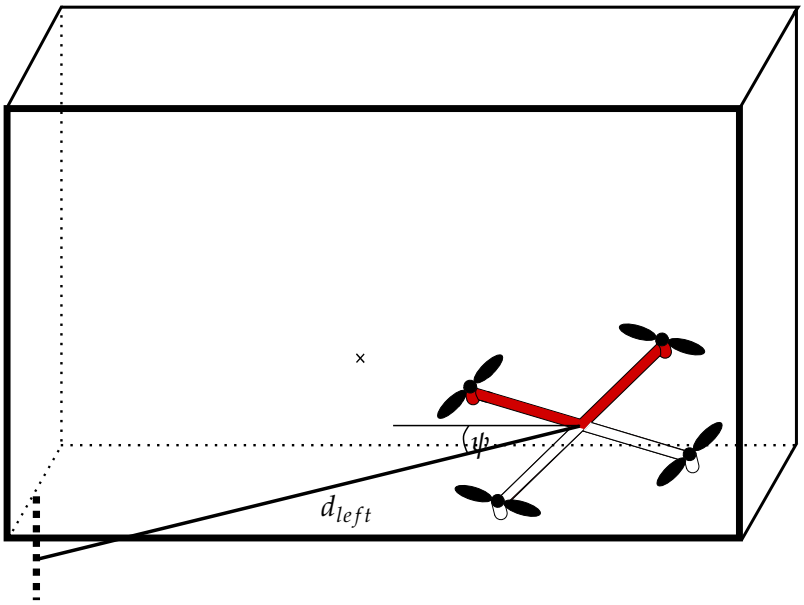


Figure 4.4. Distance to the left is not the next encountered wall

to subtract g from the acceleration in the z-axis.

Secondly, the gyroscope doesn't output values in SI-units but in different ranges depending on the value. The acceleration values lie in a range between -32768 and 32767 with $g \equiv 16384$. Accordingly, the model multiplies all acceleration values with 1670.

The angular velocities also range between -32768 and 32767 but with $250\frac{deg}{s} \equiv 32768$ and are therefore scaled with 131.072.

Since the gyroscope on the quadcopter is attached to the copter in another angle, the model has to rotate every output value corresponding to an output value of the gyroscope by 45° around the z_B axis using the rotational matrix Y_{45° from Equation 4.1.4. Taking all these adjustments into consideration, the simulation outputs the acceleration a_B in the body frame and the angular velocity ν as follows:

$$\begin{aligned} a_B &= \left(\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ g \end{pmatrix} \right) Y_{45^\circ} 1670 \\ &= \begin{pmatrix} 1670\sqrt{2}(\ddot{x} - \ddot{y}) \\ 1670\sqrt{2}(\ddot{x} + \ddot{y}) \\ 1670(\ddot{z} - g) \end{pmatrix} \end{aligned} \tag{4.1.14}$$

Table 4.2. The different constants of the quadcopter and their respective units

Parameter	Value	Unit	Parameter	Value	Unit
g	9.81	m/s^2	l	$2.980 * 10^{-6}$	–
m	0.468	kg	b	$1.140 * 10^{-7}$	–
d_r	0.225	m	I_M	$3.357 * 10^{-5}$	kg m^2
A_x	0.25	kg/s	I_{xx}	$4.856 * 10^{-3}$	kg m^2
A_y	0.25	kg/s	I_{yy}	$4.856 * 10^{-3}$	kg m^2
A_z	0.25	kg/s	I_{zz}	$8.801 * 10^{-3}$	kg m^2

$$\nu = \begin{pmatrix} p \\ q \\ r \end{pmatrix} Y_{45^\circ} 131.072 = \begin{pmatrix} 131.072\sqrt{2}(q-p) \\ 131.072\sqrt{2}(q+p) \\ 131.072(r-g) \end{pmatrix} \tag{4.1.15}$$

Lastly, the ultrasonic sensors of the quadcopter only output values in cm. Therefore the distance values are multiplied by 100 to accommodate this.

4.1.7 Physical Properties of the Quadcopter

The model presented above contains some physical constants that describe properties of the quadcopter. Since these are hard to find out without the proper equipment to for example calculate the surface area of the rotors. Therefore, these constants have been extracted from McGilvray and Tayebi [TM04]. Table 4.2 shows the values of the mass m, the distance between the rotor middle and the middle of the quadcopter d_r, the lift constant l, the drag constant b, the drag force coefficients A_x, A_y and A_z of the quadcopter in every direction as well as the inertia of the rotors I_M and the inertia around the axes I_{xx}, I_{yy} and I_{zz}. The two inertias I_{xx} and I_{yy} are identical due to the symmetrical nature of the quadcopter.

4.2 Simulating the Model

This model alone won't be enough to test the flight behavior of the quadcopter, as there is no possibility to let the program running on the Arduino interact with the model. Thus, a way to input data into the model and output the data to the Arduino is needed. KIEM provides a way to do this. KIEM uses data components to create a linear execution of the simulation. Since the model as well as the flight controller need data from the other component – and thus have to output data for each other – they both have to be producer and consumer.

This section explains the different components needed for the simulation. The components diagram in Figure 4.5 gives a short overview over these components. The simulation requires exactly one component for the flight controller and one for the model to run. The

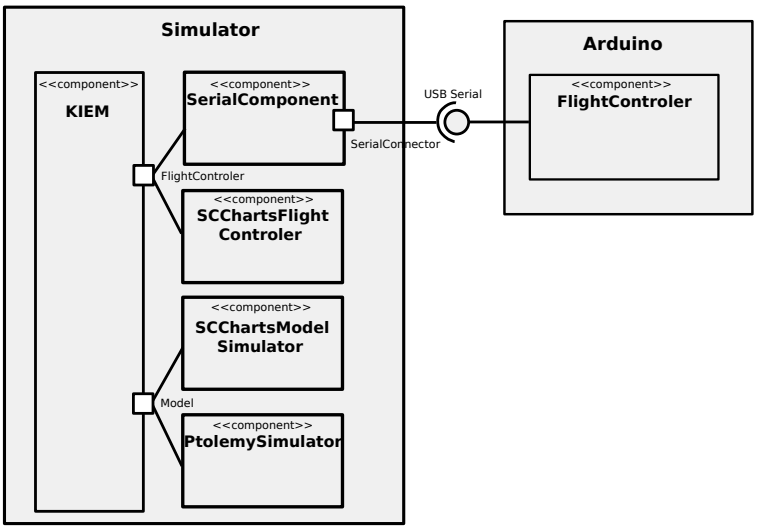


Figure 4.5. The different components of the KIEM simulation

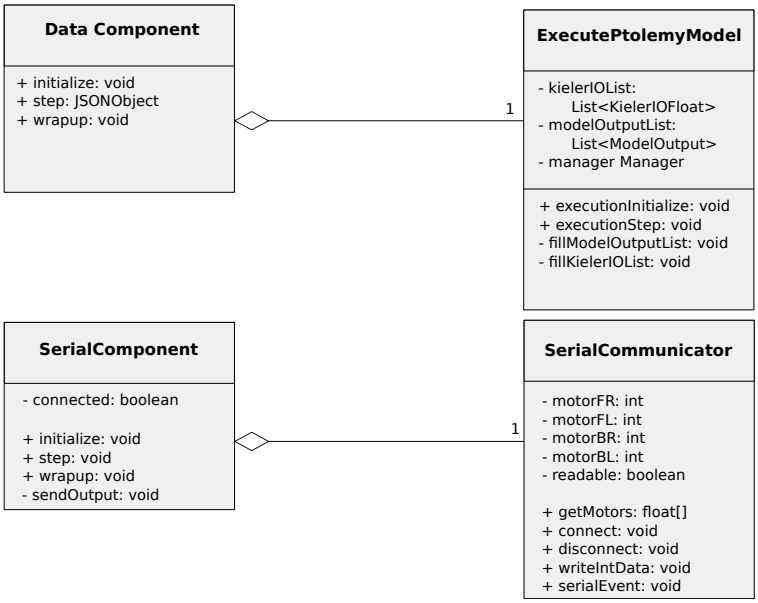


Figure 4.6. The classes of the KIEM simulation with their most important attributes and methods

class diagram in Figure 4.6 depicts the different classes of the KIEM simulation. While the *DataComponent* and *ExecutePtolemyModel* classes describe the *PtolemySimulator* component of the components diagram, is the *SerialComponent* elaborated through the *SerialComponent* and *SerialCommunicator*. The other two components were already existing before the project began and will not be explained further at this point.

4.2.1 Simulating the Flight Controller

To simulate the flight controller the user has two choices: He can either run the program on the Arduino to test the compatibility of the software and hardware or run it on the simulating PC. If the user wants to run the execution completely on the simulating PC, he can use the SCChart of the flight controller. KIEM is able to simulate any SCChart that is compilable. Running the flight controller on the Arduino however requires a serial connection to the Arduino that is realized in KIEM using the implemented *Arduino Communication* data component. This data component uses the *SerialCommunicator* java class which in turn uses JSSC to establish a serial connection between the simulation and the Arduino.

Communication Protocol

The communication follows a simple protocol. First, the Arduino sends the four motor values together with an identifying character – a number assigned to the motor in the beginning of the project that corresponds to the pin it was connected to on the Arduino. The simulation simply waits for all four motors to be transmitted. Afterwards, the simulation sends every important output of the model to the Arduino. Similarly, the simulation sends an identifying character with each output. Since the data being transmitted to the Arduino is in integers and not in bytes and the serial communication can only transmit single bytes at a time, the serial communication class cuts the integer into four bytes and sends these one by one over to the Arduino which in turn converts these bytes back to an integer. Closing the communication is a delimiter byte from the simulation signaling the Arduino that it can now continue with his calculations. On the other side, the simulation can simply continue with its computations as it does not need any more input data from the flight controller.

Alternatives to the Arduino and SCCharts

The Arduino itself is not necessary for the simulation. Both the code of the flight controller as well as the model should theoretically be testable by running everything via KIEM. The prerequisite for this is that there exists a way to simulate the flight controller in KIEM. Since the code on the Arduino that controls the flight behavior has been written with SCCharts, KIEM can easily simulate the flight controller.

If the code is not written in SCCharts, then a new data component would have to be developed. As there was code written for the flight controller in C and C++ as well for comparison purposes, a new component would have to be written to test this. This was not done in the scope of this paper due to time constraints and since the code was already tested and deemed working as of the completion of the simulation.

For further information on creating data components in KIEM, reading the diploma thesis of Motika [Mot09] is advised.

4.2.2 Simulating the Model in KIEM

To simulate Ptolemy in KIEM the data component *Simple Ptolemy Simulator*, which is created by the DataComponent class, is used. It is a component developed by Motika [Mot09] and cleaned up for this thesis that loads a Ptolemy XML file for the simulation. The Ptolemy model is then executed stepwise with every step of the KIEM execution. KIEM provides the model with input data of the motors and extracts the outputs of the simulation via the communication bus.

Similarly, the SCCharts model can be executed using the *SCCharts / SCG Simulator (C)* data component. This data component was already implemented prior to this thesis and comes with two other data components accompanying it. The *Synchronous Signal Resetter* and the *Synchronous Signal View*. While the former resets every synchronous signal at the beginning of each tick to ensure that each signal is present if and only if it is set to present in this tick, the latter creates a view showing the value of every signal at every tick. Both of these are rather unimportant for this project and could be omitted. Similar to the Ptolemy model, the SCChart is executed stepwise by KIEM and exchanges inputs and outputs with the communication bus of the simulation.

Chapter 5

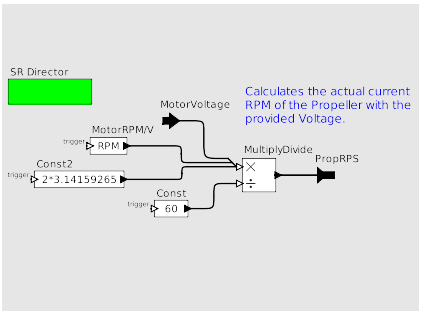
Realization

After establishing the physical properties in the previous chapter, the next section covers the realization of these models. It first explains the realization with the tool Ptolemy as well as with SCCharts. Afterwards, this chapter covers the implementations required to simulate the model using KIEM and concludes with the realization of the data exchange between the Arduino and the simulation.

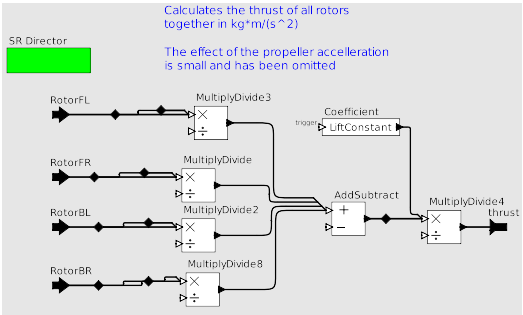
5.1 Realization with Ptolemy

The properties from Chapter 4 were subsequently modeled in Ptolemy using a continuous time director as explained in the book *System Design, Modeling and Simulation using Ptolemy II* [Pto14].

To start off, the model begins with four *KielerIO* actors as described in Section 3.1.1. These receive the input data from the KIEM simulation or, for debugging purposes, from the Ptolemy file itself. They then output these to the first composite model calculating the actual rotations per second of every rotor as these are needed to calculate thrust and torque of the copter.



(a) Calculating the rotations per second of the rotors in Ptolemy



(b) Calculating the thrust of the quadcopter in Ptolemy

Subsequently, using Equation 4.1.6 and Equation 4.1.9, the simulation computes both the mentioned thrust and torque. Calculating the thrust is fairly simple, as the model squares the revolutions per minute and adds the results of all motors together. On the other hand, differences in the axes can be spotted in the calculations for the torque. Both

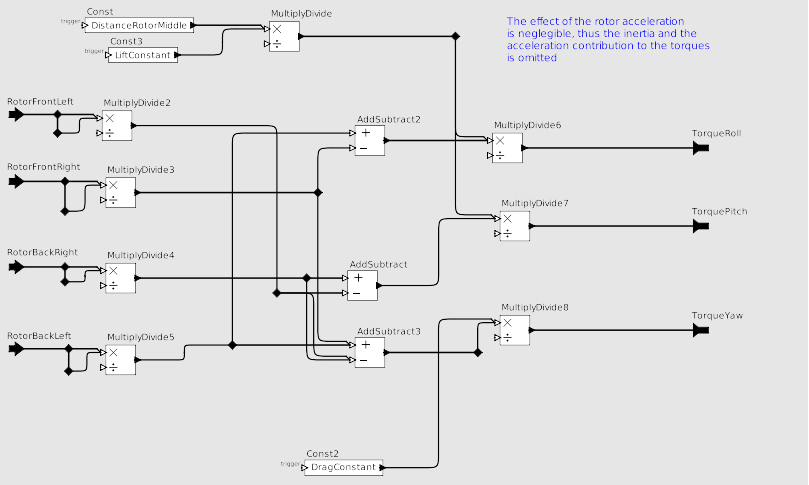


Figure 5.2. Calculating the torque of the quadcopter in Ptolemy

roll and pitch use pretty much the same logic, but yaw calculates very differently from the other two as explained in Section 4.1.3. The torque around the x and y axis is only affected by two motors respectively, since these axes are on two of the arms of the quadcopter, which contain the motors. Therefore, a change in these motors does not affect the torque around these axes. On the other hand, two of the rotors affect the torque around the z axis positively and two affect it negatively resulting in a rotation around the z_B axis. If the motors on the quadcopter have been mounted the other way around and the copter in the simulation turns clockwise while the real one turns counter-clockwise, this issue can be fixed in this part of the model.

5.1.1 Linear Accelerations, Velocities and Position in Ptolemy

With the thrust mentioned above and assuming already preexisting and correct angles, the model can now calculate the linear accelerations with Equation 4.1.7. Figure 5.3 delineates the general calculation of the accelerations. The composite models in the figure are implementations of the rotational matrix 4.1.1 applied to the thrust vector. Afterwards, the thrust, mass and gravity are factored in.

Since this is still not the real linear acceleration, the composite model in Figure 5.4 considers the aerodynamic effects on the copter. It delineates the in Equation 4.1.8 presented contexts and considers the drag force coefficients of the quadcopter in the different directions.

To accurately calculate these aerodynamic effects, the model needs to compute the linear velocity of the copter. Ptolemy already provides an *integrator actor* for integrating values over time in the continuous director. Furthermore, the simulation also calculates the

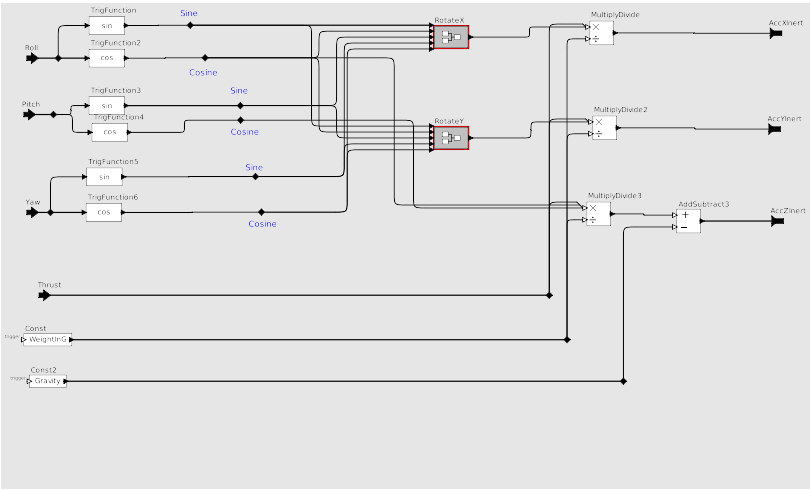


Figure 5.3. Calculating the acceleration in the inertial frame in Ptolemy

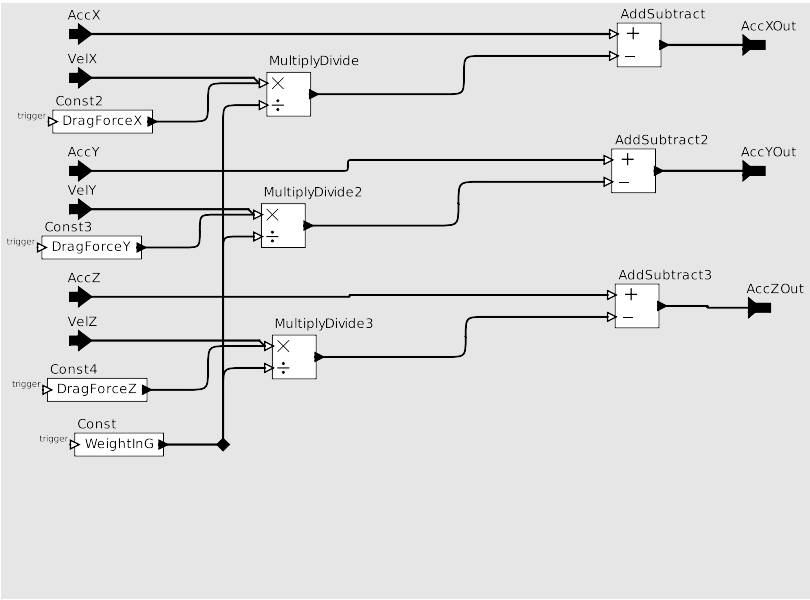


Figure 5.4. Calculating the drag affecting the quadcopter in Ptolemy

current position of the copter by integration, which it later needs for the distance sensors.

5.1.2 Angular Velocities and Angles in Ptolemy

With the torque calculated in Section 5.1 as well as the angular velocities of the rotors, the model can calculate the angular velocities of the copter. Both the angular velocities in the

body frame as well as the angles in the inertial frame are required, the former as an output to the flight controller and the latter as a variable for other calculations. Figure 5.5 shows the outline of the Ptolemy model describing the angular properties. Once again, the model makes use of the *integration actor* multiple times. Figure 5.6 shows the computations for the change in angular velocities. These need already existing values of the angular velocities as input, so the integration actors are initialized with reasonable values – for example $0\frac{m}{s}$. Equation 4.1.11 was used to create this composite model.

The modal model to the right of Figure 5.5 calculates the rotation of the angular velocities from the body frame to the inertial frame according to the rotational matrix W_η^{-1} from Equation 4.1.13. After rotating, the simulation calculates the current angles, again by integration, and then divides them modulo 360 to receive an angle between 0° and 360°. This is necessary to ensure that the modal model operates correctly. Furthermore the values of the angles have to be scaled from degree to radians, as the sine, cosine, and tangent actors in Ptolemy use angles in radians.

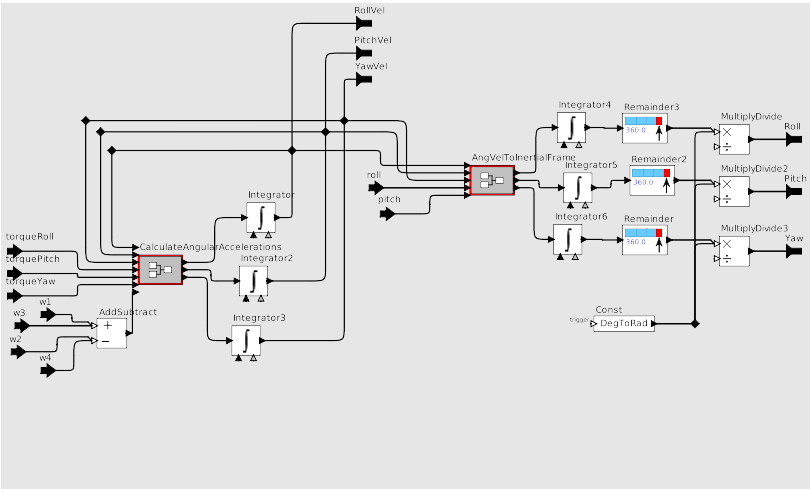


Figure 5.5. Calculating the angles in Ptolemy

5.1.3 Distance Calculation in Ptolemy

The distance calculation in Ptolemy uses a modal model – a state machine – with four states, one for each general direction the copter could be facing. Every state contains a refinement, each describing one column of the equations from Table 4.1 appertaining to the direction. Figure 5.7 shows this modal model while Figure 5.8 depicts one representative refinement.

Since the times for a tick are very short with just a few milliseconds in general, it is not important for the states to have transitions to the opposing state. For example, the state

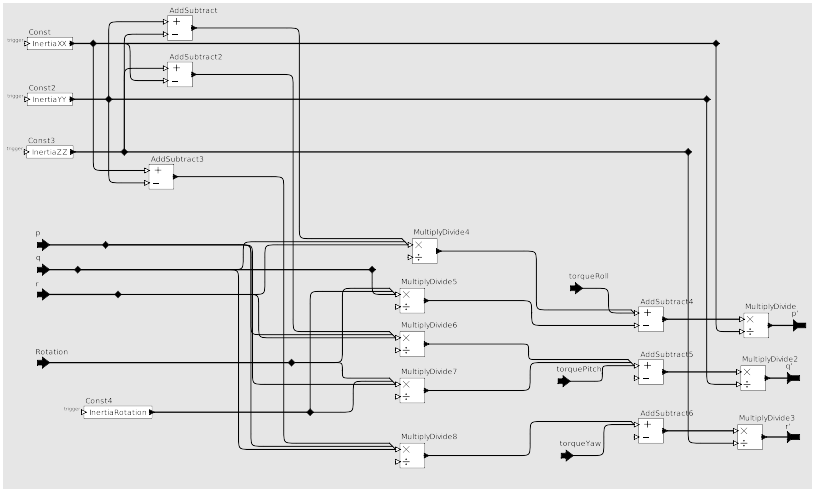


Figure 5.6. Calculating the angular accelerations in Ptolemy

facingLeft does not need a transition to *facingRight*, because such a sudden change in one tick is very unlikely if not impossible. In case this happens, the state between facingLeft and facingRight would merely been active for one tick. This means that its calculations are also only relevant for one tick. This would not create a lot of problems since the distance measurement smooths singular outliers [Mac15].

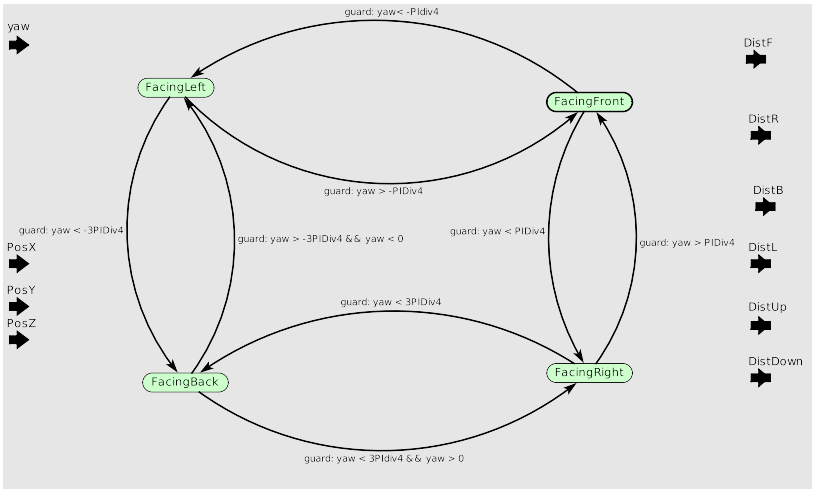


Figure 5.7. The modal model describing the current direction of the quadcopter in Ptolemy

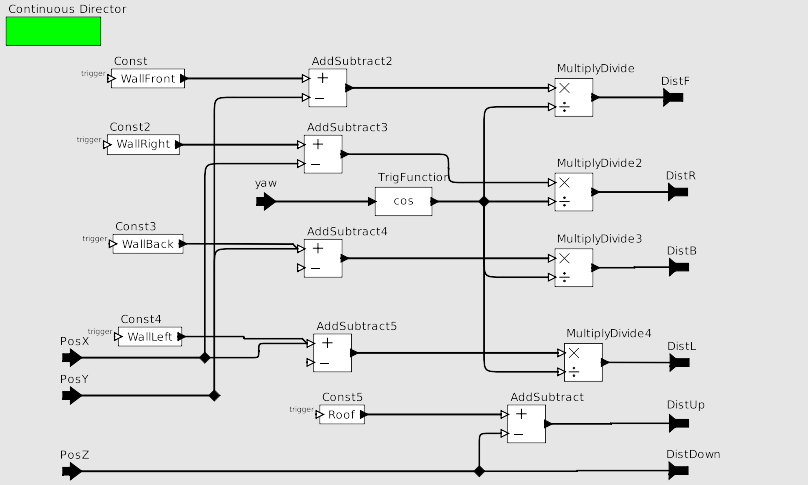


Figure 5.8. Calculating the distances to the walls in Ptolemy

5.1.4 Adjusting the Output Values in Ptolemy

Finally, to adjust the output values, the simulation first rotates the acceleration values back into the body frame, rotates both the angular velocities and the acceleration by 45° and then simply scales each of the output variables with the corresponding constants mentioned in Section 4.1.6.

5.2 Realization with SCCharts

Similar to the section above, this section will deal with the implementation of the model with the help of data-flow in SCCharts. Unlike Ptolemy, everything in SCCharts is written in the SCT language and then visualized by a layout algorithm and is not created in a graphical editor. Thus, the creator only has complete freedom over the content and not over the layout of the model. Since the code snippets describing the model are fairly simple as they are pretty much the equations from Chapter 4, the figures in this chapter will be of the visualization of the model.

The different parts of the model are realized in multiple regions. Due to the sequentially constructive MoC of SCCharts the executions of the regions are scheduled in a way such that all variables are first initialized, then updated and lastly read from [HMA+13]. Therefore, the execution order of the code generated by SCCharts is determined by the scheduler. The order of execution is not the same as the order of the visualization.

Due to the complexity of the topic, this section is ordered mostly the same as Chapter 4 and Section 5.1. Just like in Ptolemy, the model first starts with the calculation for the propellers and the thrust. Contrary to the model in Ptolemy, this model calculates the

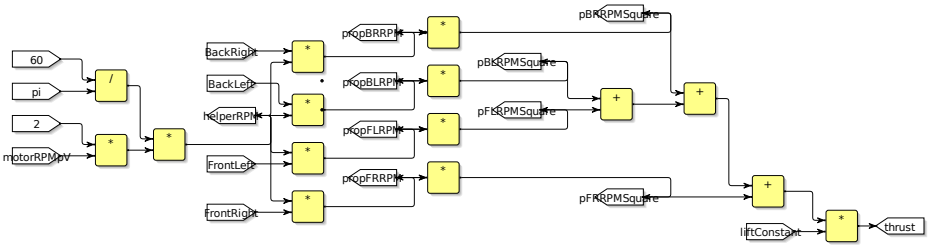


Figure 5.9. Calculating the thrust in SCCharts

torque in another region. This has no specific reason, other then the thematic relation. Figure 5.9 shows these calculations. Since the calculation is done in one model, there is much less redundant computation than in Ptolemy. The picture further shows one of the main problems in data-flow in SCCharts right now: Many helper variables such as the *propBLRPM* variable do not need to receive *markings* in the visualization. Furthermore, these markings are often times not large enough to fit the entire name of a variable.

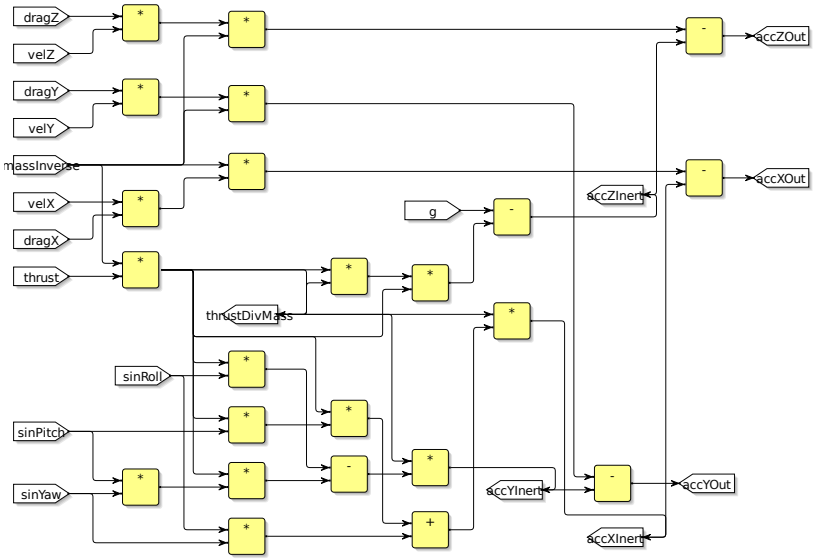


Figure 5.10. Calculating the accelerations in SCCharts

5.2.1 Linear Accelerations, Velocities and Position in SCCharts

Figure 5.10 and Figure 5.11 delineate the calculations for the linear accelerations of the quadcopter. They encompass all the necessary calculations including the computation of the drag counteracting the acceleration as depicted in the top of the model in Figure 5.10 as well as scaling the output to values that can be read by the Arduino. Figure 5.11 depicts

the rotation back into the body frame on the left of the model and calculates the actual outputs to the Arduino in the top right by rotating the x and y acceleration values by 45° as described in Equation 4.1.4 and then scaling the outputs as described in Section 4.1.6.

On the other hand, Figure Figure 5.12 visualizes the calculations for the velocity and position. In the current version of the model, referenced SCCharts is not used and the integration is executed in the main model. This leads to some redundancy and takes away some clarity from the visualization.

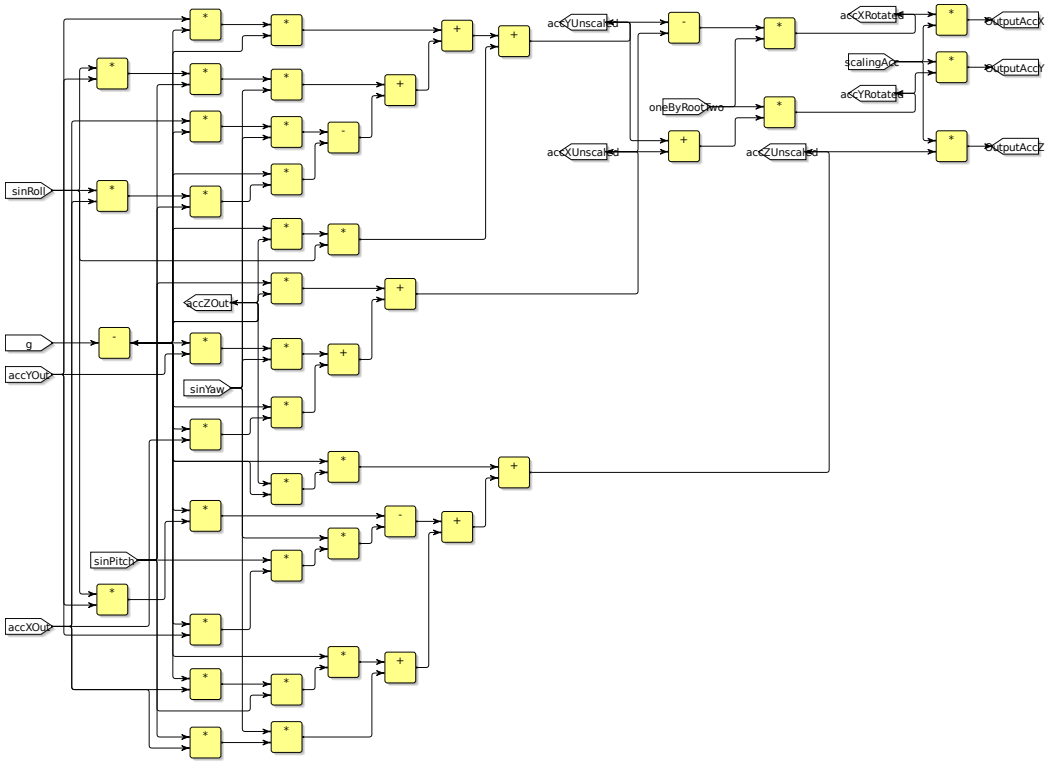


Figure 5.11. Calculating the output accelerations in SCCharts

Integration in SCCharts

SCCharts itself does not have an integrate function. Thus, a SCCharts model was developed using the same integrating method as Ptolemy to ensure that both simulations will be as similar as possible. This method is the Riemann summation using the trapezoidal rule. If the intervals of the sum are small enough, the sum approaches the Riemann integral [SG78] and can therefore be used to approximate the integral [TF96]. With this method the model

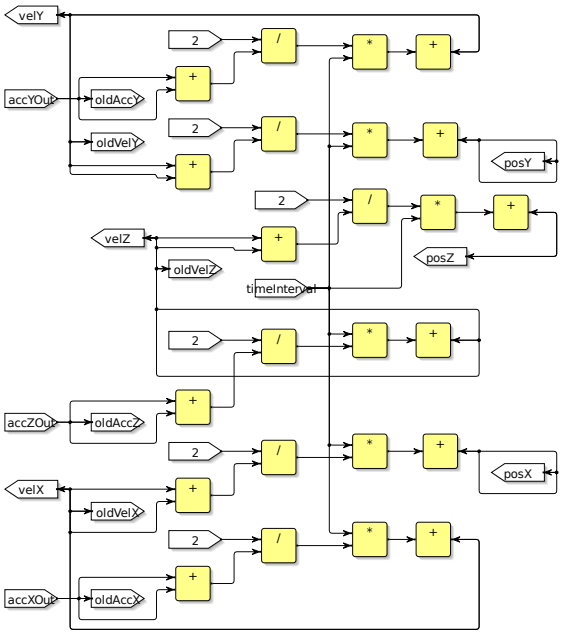


Figure 5.12. Calculating the velocity and the position in SCCharts

calculates the integral x of $\dot{x}$ at time n as follows:

$$x[n] = x[n-1] + \frac{1}{2}(\dot{x}[n]\dot{x}[n-1])\Delta t$$

Since the integration was used multiple times in the model, referenced SCCharts could have been used instead of the current approach.

5.2.2 Angular Velocities and Angles in SCCharts

As mentioned, the torque of the quadcopter is calculated at another place in the model in SCCharts then in Ptolemy. Here, the torque is calculated together with the rest of the angular calculations. Figure 5.13 and 5.14 together describe these calculations. The layouts of these two regions are sadly confusing and hard to follow. Just like above, they contain the integration themselves and not as a referenced SCChart, complicating the layout even more. Figure 5.13 contains the information about the angular velocities p, q and r while Figure 5.14 visualizes the calculation of the angles. Not depicted is the computations of the trigonometric functions of ϕ, θ and ψ since data-flow does not visualize these currently.

In the bottom left of Figure 5.13 are the calculations for the torque as described in Equation 4.1.9. The top however depicts the calculations for the angular velocities. Since they are in a loop-back due to the integration, the data-flow is hard to follow.

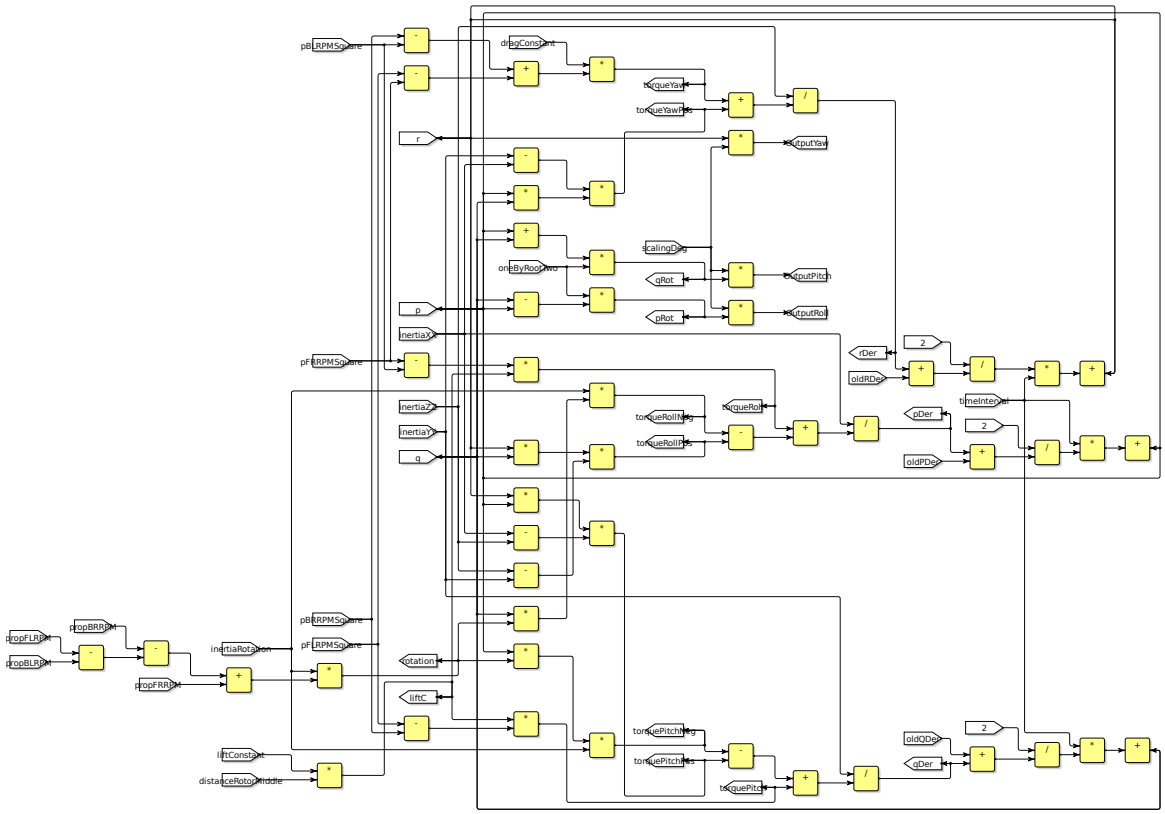


Figure 5.13. Calculating the angular velocities in SCCharts

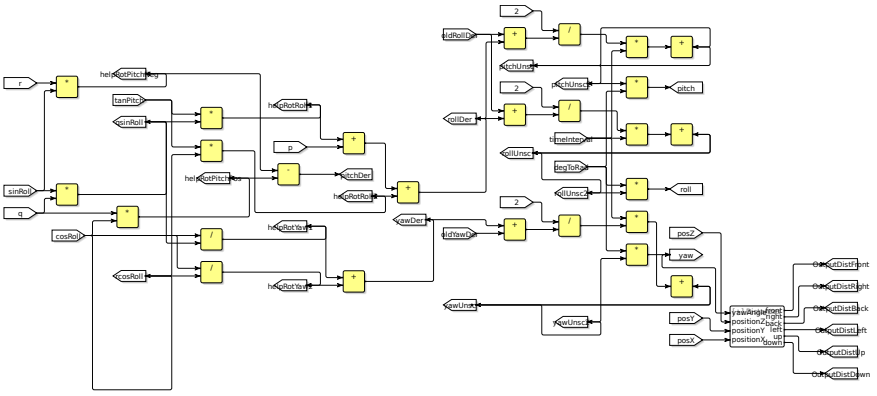


Figure 5.14. Calculating the current angles in SCCharts

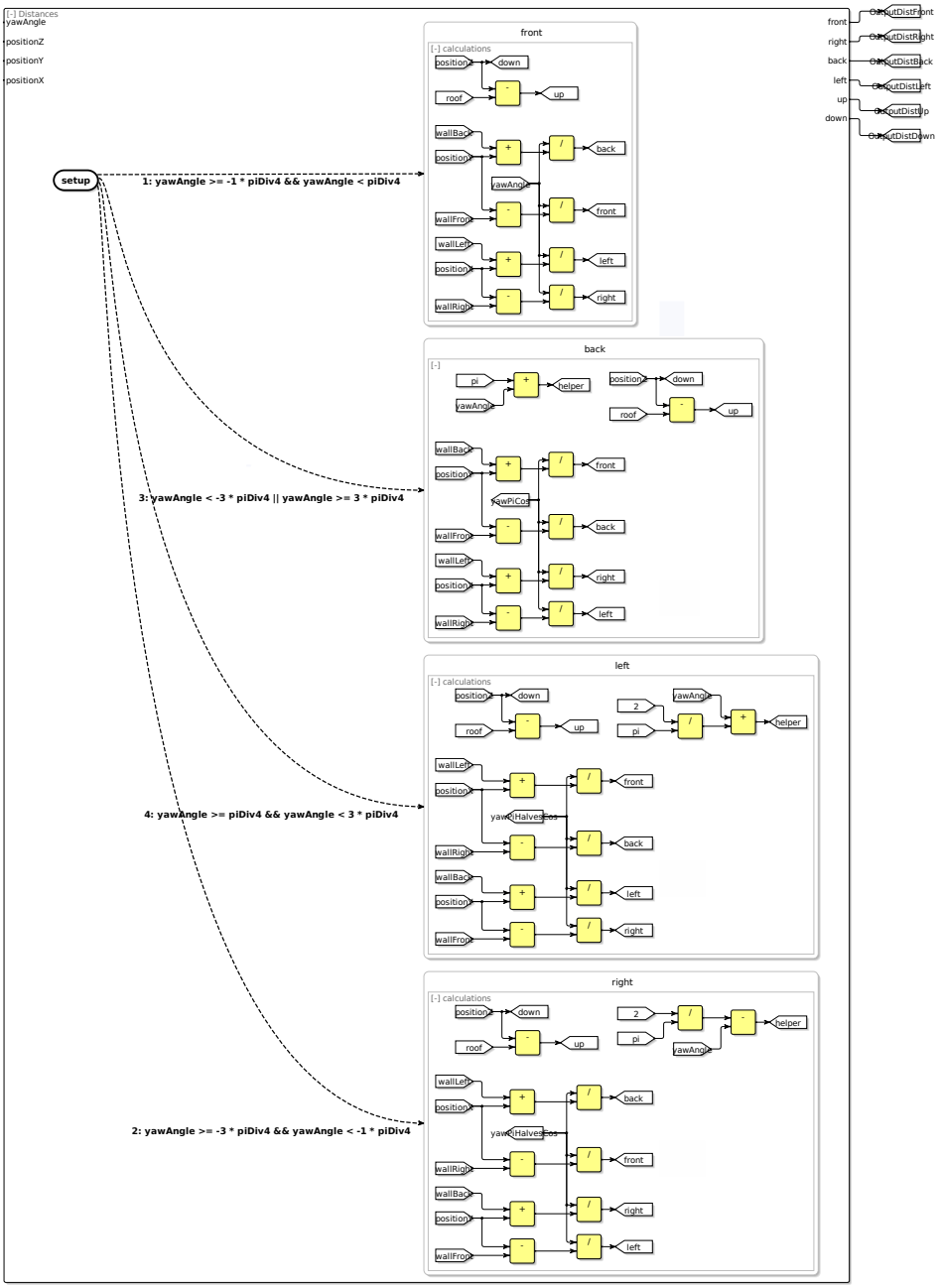


Figure 5.15. Calculating the distances between the walls and the quadcopter in SCCharts

5.2.3 Distance Calculations in SCCharts

The nested model for the distance calculations can already be seen in Figure 5.14. This SCChart, as visualized in Figure 5.15, does the same as the modal model in Ptolemy in

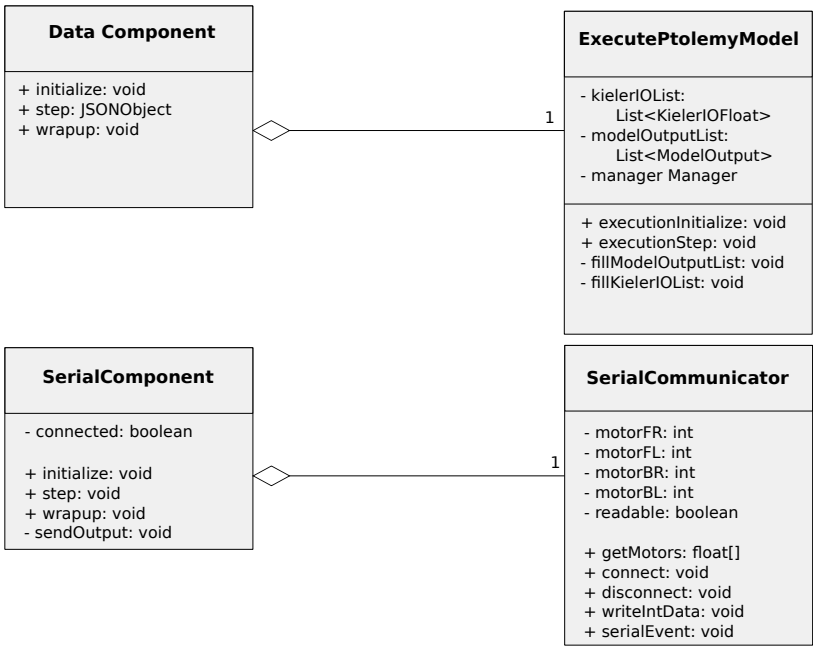


Figure 5.16. The classes of the KIEM simulation with their most important attributes and methods

Figure 5.8 and calculates the distances to the different walls yet does it slightly differently. Contrary to Ptolemy, the model does not memorize the current state it is in but has to calculate the direction the quadcopter faces in every tick by first entering the nested SCChart in the state *setup*. The control-flow then enters one of the next states via an *immediate transition* depending on the current yaw angle ψ.

5.3 Simulating the Model

As explained in Section 4.2, the model cannot simulate the behavior of the quadcopter without help. Therefore, this section explains the required software to simulate the model. First, this section will cover the data component for the Ptolemy model. Since the SCCharts data component was unchanged, it is not important to take a closer look at this component. Following that, there will be a short paragraph about the serial communication components on both the side of the simulation as well as on the side of the Arduino.

5.3.1 Ptolemy Data Component

The *Simple Ptolemy Simulation* is a data component developed by Motika [Mot09] that was cleaned up for the purpose of this paper. The functionality of this data component is explained in detail in his thesis. Therefore, this section only summarizes this data

```
1  private void fillKielerIOList(final List<KielerIOFloat> kielerIOListParam,
2            final List<InstantiableNamedObj> children) {
3
4    for (Object child : children) {
5      if (child instanceof CompositeActor) {
6        CompositeActor compositeActor = (CompositeActor) child;
7        fillKielerIOList(kielerIOListParam, compositeActor.entityList());
8      }
9      if(child instanceof KielerIOFloat) {
10       kielerIOListParam.add((KielerIOFloat) child);
11     }
12     if (child instanceof ModalModel) {
13       ModalModel modalModel = (ModalModel) child;
14       fillKielerIOList(kielerIOListParam, modalModel.entityList());
15     }
16     if (child instanceof ModalController) {
17       ModalController modalController = (ModalController) child;
18       fillKielerIOList(kielerIOListParam, modalController.entityList());
19     }
20   }
21 }
```

Listing 5.1. Loading the input and outputs of the Ptolemy model

component. For further information, reading the thesis of Motika is advised. The class diagram in Figure 5.16 is the same as the one in Section 4.2 included again for easier reading.

The initialization of the DataComponent begins with loading the XML-file of the model and parsing this model into a string that is readable for the program. Subsequently, the execution needs fills a list of every object in the model so it can find every input, output and nested models. Every *KielerIO* actor or, adapted for this thesis, every *KielerIOFloat* actor is considered an input variable and every *MonitorValue* actor an output variable. The method to load inputs is described in more detail in Listing 5.1. It searches the current model for every nested model, recursively calling the search for more inputs in every nested model. If it finds a *KielerIOFloat*, it adds it to the list of input actors which are needed later. The method to load outputs does the same without searching for outputs in nested models. These inputs and outputs are then stored in the *kielerIOList* and the *modelOutputList* respectively. Afterwards, the initialization starts the simulation.

The most important part of the component is the *step* function. Listing 5.2 depicts that the step starts with exporting the input data from the simulation, the *jSONObject*, to the model and then executes one step of the model, which is shown in Listing 5.3 in more detail. Since Vergil is merely a graphical UI of Ptolemy and Ptolemy can run alone without

```
1  public JSONObject step(JSONObject jSONObject) throws KiemExecutionException {
2    JSONObject returnObj = new JSONObject();
3    try {
4    model.setData(jSONObject);
5    model.executionStep();
6    } catch (Exception e) {
7     // Catch Exception
8    }
9    List<ModelOutput> outputs = model.getModelOutputList();
10   for (ModelOutput output : outputs) {
11     String signalName = output.getName();
12     boolean present = output.isPresent();
13     Double value = output.getValue();
14     try {
15       JSONObject signalObject = JSONSignalValues.newValue(value, present);
16       try {
17         returnObj.accumulate(signalName, signalObject);
18       } catch (Exception e) {
19         // Catch Exception
20       }
21     } catch (JSONException e) {
22       // Catch Exception
23     }
24   }
25   return returnObj;
26 }
```

Listing 5.2. Starting a step of the Ptolemy model

this UI, this step is executed without the need of the visualization provided by Vergil. After the step, the component receives the values from every output and writes these values as well as the corresponding name of the actor into a JavaScript Object Notation (JSON) object, a language independent data format often used to transmit information between different parts of a system. In KIEM, JSON is used to transmit information between the different data components. All JSON objects are now accumulated into one JSON object and then saved in the simulation so they can be used by other parts of the KIEM execution.

The step execution in Listing 5.3 is the one finally executing the step in the actual Ptolemy model. First it changes the present flag of all output signals to false as to overwrite the changes from the previous tick. Afterwards, the system gets the value of every input variable in the kielerIOList by extracting the value of the JSON object with the same name as the input variable. They are subsequently written into the KielerIO object corresponding to the input variable and the Ptolemy manager executes a step of the model in Ptolemy.

Once the user ends the simulation, the *wrapup* function is called by KIEM. This function

```
1  public synchronized void executionStep() throws KiemExecutionException {
2    try {
3      for (ModelOutput modelOutput : modelOutputList) {
4        modelOutput.present = false;
5      }
6      float value = 0;
7      for (KielerIOFloat kielerIO : kielerIOList) {
8        String signalName = kielerIO.getSignalName();
9        kielerIO.setPresent(isSignalPresent(signalName));
10       Object objVal = getSignalValue(signalName);
11       if (objVal instanceof Float) {
12         value = (Float) objVal;
13       } else if (objVal instanceof Double) {
14         value = ((Double) objVal).floatValue();
15       } else if (objVal instanceof Integer) {
16         value = ((Integer) objVal).floatValue();
17       }
18       kielerIO.setValue(value);
19     }
20     manager.iterate();
21
22   } catch (KernelException e) {
23     //Catch Exception
24   }
25 }
```

Listing 5.3. Loading the inputs during a step and executing the step

however merely stops the execution of the model and unloads the XML file.

5.3.2 Communication between the Simulation and the Quadcopter

To exchange data between the flight controller and the model, the simulation requires a communication component. This component requires the protocol from Section 4.2.1 which it adheres to as well as an implementation both on the side of the simulation as well as on the side of the flight controller.

Serial Communication Data Component

The *Arduino Communication* data component offers the serial communication between the models and the Arduino. It utilizes the *SerialCommunication* java class as the data component as well as the *SerialCommunicator* java class using simple JSSC commands. Using the same structure as all data components, the initialization begins with connecting the simulation via a serial port to the Arduino. The *initialize* function of the data component

```
public JSONObject step(JSONObject jSONObject) throws KiemExecutionException {
  JSONObject returnObj = new JSONObject();
  JSONObject[] signalObject = new JSONObject[4];
  while(!com.isReadable()) {
    try {
      Thread.sleep(5);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  float[] motors = com.getMotors();
  try {
    for(int i = 0; i < 4 ; i++) {
      signalObject[i] = JSONSignalValues.newValue(motors[i], true);
    }
    returnObj.accumulate("FrontLeft", signalObject[0]);
    returnObj.accumulate("BackRight", signalObject[1]);
    returnObj.accumulate("FrontRight", signalObject[2]);
    returnObj.accumulate("BackLeft", signalObject[3]);
  } catch (JSONException e) {
    // Catch Exception
  }
  try {
    JSONObject xAcc  = (JSONObject) jSONObject.get("OutputAccX");
    JSONObject yAcc  = (JSONObject) jSONObject.get("OutputAccY");
    //[...]
    com.writeDelimiter();
  } catch (JSONException e1) {
    // Catch Exception
  }
  return returnObj;
}
```

Listing 5.4. Executing a step in the Serial Component, transferring data between the simulation and the Arduino

calls the *connect* method of the SerialCommunicator class, which opens the port to the first connected serial port. Since we assume that only the Arduino is connected to the simulating PC, this should be sufficient. Listing 5.4 then depicts how a step is executed in the serial component. The component first waits for the motor outputs from the Arduino, if they haven't already arrived. As soon as the component receives the outputs, it saves them to the JSON object of the simulation that is loaded by the simulation as described in Section 5.3.1. Afterwards, it sends all the outputs of the model by acquiring the corresponding JSON objects from the simulation and writing their values to the serial port together with a letter

```
1 serialPort.writeBytes(name.getBytes());
2 for(int n = 0; n < 4; n++) {
3   serialPort.writeByte((byte) (data & 0xFF));
4   data >>= 8;
5 }
```

Listing 5.5. Shifting an integer so it can be sent byte for byte

identifying the output so that the Arduino can save the values to the correct variables. At the end of the transmission the controller sends a delimiter byte containing only ones to the Arduino.

Since JSSC can only send single bytes in a format that the Arduino is able to read, the communicator has to send integers as four bytes to the Arduino, which in turn has to write these bytewise into an integer. Listing 5.5 shows that this was done using the bitwise shift command in Java. Masking the value so that only the first eight bits are written into the byte is not necessary, yet helps to understand the process of the transmission.

Serial Communication on the Arduino

On the other end of the communication is the Arduino. The libraries on the Arduino already implement serial communication and thus only small adjustments were necessary.

Every tick on the Arduino, the flight controller calls the *setMotor* function for every Motor and the *readValue* function to get all required inputs. Listing 5.6 depicts how a motor value is written to the serial port and shows with an example how the readValue function operates. The *setMotor* functions simply write the name of the motor and the corresponding value as a byte onto the serial port. Since the motor values of the flight controller never exceed 255, a byte is enough to transfer the information of the motors. On the receiving end in the readValue function, the Arduino loops as long as it has not received the delimiter byte. If the Arduino receives a letter identifying a variable, the following four bytes are written to this variable. If the delimiter byte has been received and not all values have been overwritten, these values stay the same as in the last tick. While it is reasonable to argue that the Arduino should re-request data it has not received, this way we can simulate behavior occurring in real environments. In our tests we often times realized that sometimes some input data from the gyroscope are lost and the flight controller continues operating with the old values.

It is important to note that this synchronization determines that one tick in the model equals one tick on the Arduino. While this might not be the best solution, it disallows very asynchronous behavior. Thus, the simulation cannot do multiple steps while the Arduino is too slow to react making him crash or allows the Arduino to do multiple steps during one instance of the environment causing outputs to be lost.

```
void SimulationConnector::setMotor0(int throttle) {
    Serial.write('9');
    Serial.write(number);
}

//[...] Other transmission functions for sending motor values

void SimulationConnector::readValues(int *serialValPtr, float *yprPtr,
                                     int *accPtr, int *distancesPtr) {
  int inChar = 0;
  int value = 0;
  boolean light = false;
  delay(200);
  digitalWrite(13, LOW);

  int i = 0;

  while(!ended) {
    delay(5);

    if (Serial.available()) {
      int inChar = Serial.read();
      value = 0;

      if(inChar == 255) {
        ended = true;
      } else {
        while (Serial.available() < 4) {
          delay(5);
        }
      }

      //Acceleration in x-axis
      if(inChar == 'X') {
        value = Serial.read();
        value = value + (Serial.read() << 8);
        value = value + (Serial.read() << 16);
        value = value + (Serial.read() << 24);
        accptr[0] = value;
      }
      //[...] Other received variables
    }
  }
  ended = false;
}
```

Listing 5.6. Examples of transmitting and receiving functions of the Arduino

Chapter 6

Evaluation

This chapter evaluates and compares the two different approaches to simulate the quadcopter. First, it gives a short overview over the results of the simulation using the flight controller. Afterwards, it compares the different implementations, the differences in the calculations as well as the respective advantages and disadvantages of the two approaches. Lastly, it explains problems encountered in the making of the simulation.

It is important to note once again that the simulation uses constants describing another quadcopter. The results of this simulation sadly do not guarantee that a quadcopter running on the tested software is able to fly or not as it could be unbalanced like the one built as part of the project.

6.1 Results of the Simulation

There are a lot of problems with the simulation. Sadly, due to time constraints towards the end of the project the simulation could not be properly tested in the context of the SCCharts flight controller. Since the flight controller has a lot of adjustments made to fit the hardware and the collision avoidance and the flight controller were not implemented in one SCChart but in two different SCCharts, it could not properly be tested. Regardless, the model was tested with the flight controller alone without the collision avoidance.

6.1.1 Problems with the simulation

After adjusting the names of the variables and removing constraints on the controller that were made due to hardware constraints of the quadcopter, the flight controller would output -2147483648, the largest negative value of an integer, on every motor, regardless of the input of the simulation. Testing the flight controller alone results in the same output values. After some more testing and changing the time input variable to be present at every tick, reasonable values were produced by the simulation. Making the input time to a constant integer or locking it in KIEM to output every tick did not work however. Therefore, the input had to be manually set to present at every other tick of the simulation.

Even after these adjustments, the simulation could not produce realistic values. While this could be due to the fact that the copter in the model does have different physical properties to the copter that the flight controller was written for, further investigations

showed that a flaw in the calculations is more likely. Tests without the flight controller concluded that the acceleration values seemed too high while the angular velocities in comparison seemed very low.

Another problem is the input to the motors of the quadcopter. Since the component outputting the voltage to the motors is outputting them with a Pulse Width Module and their data sheets are non existent, it is impossible to find out how much voltage is provided at which value of the motor output of the flight controller. Therefore, it is hard to properly adapt the simulation to a real quadcopter. Other work on the physical properties of quadcopters bypassed this issue by inputting the actual measured angular velocity of the rotors like Höger [Hög14] or using a feedback control with a current and a desired state like Luukkonen [Luu11].

6.2 Comparison between Ptolemy and SCCharts

As evident from the previous chapter, Ptolemy and SCCharts have their differences. While some of them like the design of the actors are pretty apparent just from looking at the figures in Chapter 5, many more become obvious using the two tools. Not only are there implementation differences, there are also slight differences in the calculation.

6.2.1 Implementation Differences

The first and most glaring difference between Ptolemy and SCCharts is the method of creating a model. While in Ptolemy the user builds the model using various building blocks such as the actors and connecting these with each other mostly using drag and drop, he writes the equations in SCCharts directly in SCT. Furthermore, in SCCharts the layout of the displayed model is completely automatically created, whereas in Ptolemy the user himself has control over the layout and can either choose to use an automatically generated layout or create the layout himself. Those two aspects create two very different ways to designing a model. The method used in SCCharts makes it much easier for new users. They can simply create their model by writing down the equations they want to simulate. While the user has to know some commands to create a new model, if he just wants to create a small and quick data-flow simulation, he doesn't need to know much about these commands. On the other hand, Ptolemy is not very difficult to understand either. Simple mathematical equations most often only use addition and multiplication or sometimes integration or trigonometric functions. These are easy to use, if sometimes hard to find, in Ptolemy. The only negative aspect of Ptolemy concerning new and inexperienced users is the multitude of different directors the user might not necessarily know. As already mentioned before SCCharts does not have built in integration and the modeler has to fall back on hostcode if he wants to use trigonometric functions. Concerning the integration, this necessitates that the modeler has to create a method to integrate himself.

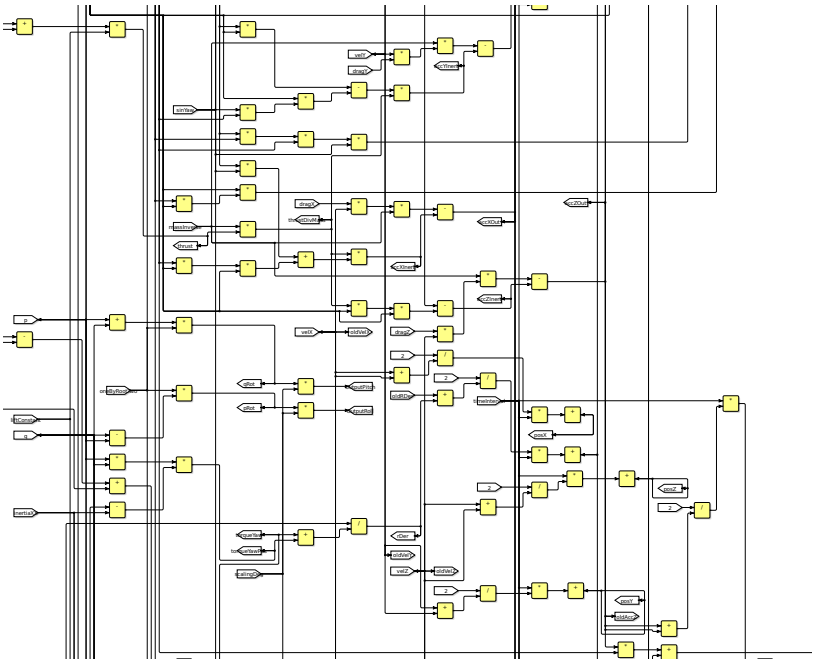


Figure 6.1. A very cluttered and unclear part of the model in SCCharts

Since SCCharts creates the layout for the user, he does not need to focus on creating a good layout. Sadly, the data-flow layout in SCCharts is often times not very clear, especially for large models. The modeler can choose to create one big model that contains everything maintaining all connections like in Figure 6.1 or create multiple small regions like in Figure 6.2. While the second option maintains a clearer layout, it loses some relations between some components. For example, in Figure 6.2, the actor 'AccZOut' in the top left of the region 'CalculateAccelerations' is an output that is required as an input in the region 'CalculateVelocityAndPosition' on the left. A visually better option as depicted in Figure 6.3 is to create nested SCCharts containing a single state that then contains the equations in a data-flow environment. The picture shows two parts of the model, one opened and another one closed. Sadly, the inputs are still not completely connected to the actual actors. While this also creates a lot of work for the modeler it is probably the best option as unnecessary information can be hidden and a lot of clutter can be removed from the model.

On the other hand, Ptolemy users can create their own vision of the model. Actors and composite models can be moved around manually and a good modeler can create a model that is clear and can be understood very easily. This of course can take up a lot of time and effort the modeler does not necessarily have to expend. Alternatively, Ptolemy offers the possibility to layout the model automatically as mentioned above. This automatic layout is generated by the KLay Layered algorithm from the KIELER project [SSH14]. SCCharts

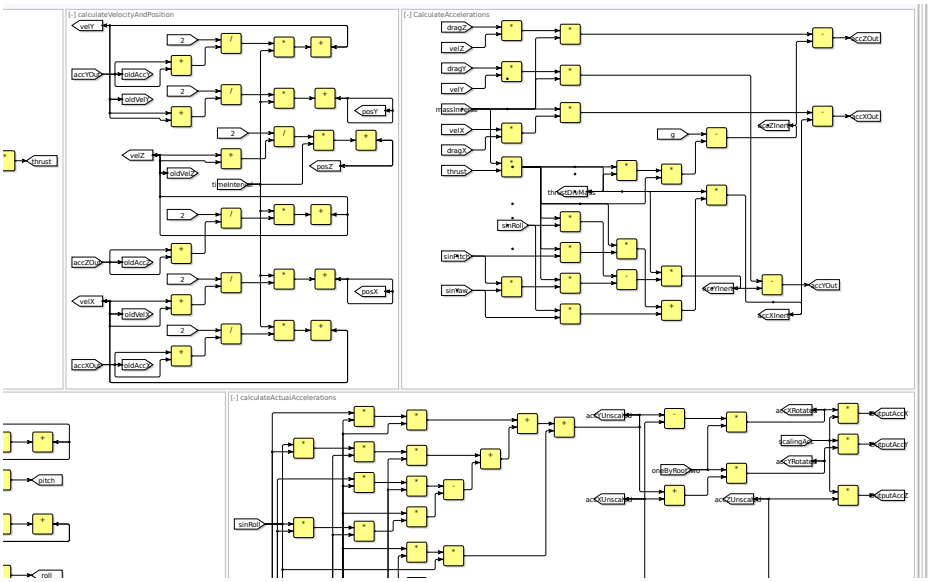


Figure 6.2. Data-flow divided into different regions

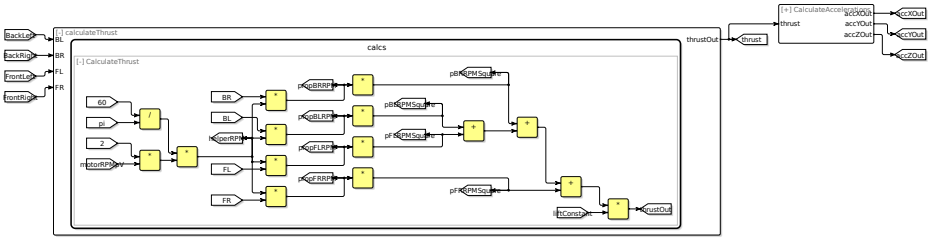


Figure 6.3. Two different data-flow regions in nested SCCharts with variables that carry over

however does not offer the possibility to manually edit the layout of a model.

Another important aspect of modeling is reusing parts of a model. Instead of using a copy and paste approach, it is often of advantage to create a way to reuse a part of the model in another place, while still maintaining the information that this part of the model is a copy of the original. This way, fixing a mistake in the original can also fix the mistake in the copy. Otherwise, the modeler might have to repair multiple chunks of the model. This kind of feature is implemented both in Ptolemy by converting an actor to a class and then creating instances of this class as well as in SCCharts via referenced SCCharts.

6.2.2 Differences in Calculations

One big difference in the calculation could be the order of computation. In data-flow in SCCharts it is pretty clear which computation comes before which computation and

even if variables occur in multiple different regions, the order of computation is given by the sequentially constructive MoC. This MoC states that in one tick a variable should be initialized first, then updated and lastly read. On the other hand, the order of computation in Ptolemy depends on the director used in the model. The continuous director used mostly in the model of this thesis converts all actors into one big equation and calculates the values of each variable with the help of a *solver*. The book *System Design, Modeling and Simulation using Ptolemy II* [Pto14] presents the different solvers as well as the computation methods for other directors. Ultimately this should not change much in the behavior of the quadcopter, as one tick should cause only small increments in the angles and thus only small differences in the acceleration values, yet can change the calculations in the simulation alone a lot.

Another difference between the two models could be the integration. Yet, since the integration used in SCCharts in this thesis was modeled after the integration method in Ptolemy, there should be no noticeable difference in the calculation of the integration. The only factor impacting different outcomes is the time of a tick. Since the time of a tick can vary wildly in Ptolemy with the continuous director, the results of the integrations can be very different from the SCCharts model as this model uses a constant time interval.

6.3 Encountered Problems

In the creation of the models I have encountered a lot of problems. Most of these problems could be solved directly or with a little bit of a workaround. Sometimes these problems were more severe and required – or will require – attention from the creators of the software. In the following I will describe these problems and how I solved them or how and why I could not solve them.

6.3.1 Problems Encountered using Ptolemy

Ptolemy is a fairly complex program. Since I already had some experience working with Ptolemy, I luckily did not have to get used to the peculiarities of the program. These peculiarities include the inability to attach the output of one actor to the inputs of multiple other actors. To do this, the user has to create a so called *relation*, a sort of anchor that a connection between elements has to go through – weirdly enough a relation is both a connection between actors and/or models and the afore mentioned anchor. These relations can then connect to multiple input ports of other actors or models. The automatic layout for relations is furthermore sometimes a bit finicky and can go through places the user does not want it to go through. Therefore, I made extensive use of the anchor relations that force the relation along a more or less fixed path. Using the automatic layout generation might have solved these problems but I personally prefer to create the layout myself in these projects because I can design the layout close to a mental map of the project I have.

The automatic layout generation of Ptolemy also currently does not consider annotations and comments. These will be arranged at the top of a model with a automatic layout. On the other hand, creating and maintaining a clear design is often fairly difficult, especially in some of the composite models that calculate the rotations. Fixing these layouts would be time consuming and might not necessarily yield much better results.

Another big problem encountered was the pause function. Ptolemy itself has its own run function. With it the user can see the change of outputs in real time. Pausing the run function causes the model to stop calculating. Still, the system clock continues. This causes the integrator values to jump after continuing, since they calculate their values according to the system time. Luckily, this does not occur when using KIEM to simulate Ptolemy. This presumably has to do with the properties of the director as some of these are discarded when KIEM runs the model.

Lastly, the Ptolemy data component for KIEM uses a dependency on an old Ptolemy Java project correlating to version 8.0.1 of Ptolemy while the simulation runs on version 10.0.1. Furthermore, there was no newer version of the Java project available to create a new dependency. Luckily, this did not create big problems. The only noteworthy problem that occurred was that in the XML-file every mention of *gui.MonitorValue* had to be renamed to *MonitorValue*. Furthermore, for KIEM to recognize the *MonitorValue* actors as an output, the user has to add a property to the actor called *signal name*. The value of that property is then displayed in the KIEM simulation. This was already in the original approach. Otherwise, the functionality of version 8.0.1 was sufficient for this thesis.

6.3.2 Problems Encountered using SCCharts

Before going into the problems I have encountered using SCCharts, it is important to once again stress that data-flow in SCCharts is a recent development emerged from the diploma thesis of Umland [Uml15]. As such, many of these problems are attributable to the general lack of polish and the unfinished nature of the project and were fixed in a very short time. As such, many of these problems would not be encountered by a new user now.

Fixed Problems

At the beginning of the project, many of the models created were compiling properly to code using the SCCharts synthesis. Yet, when looking at the generated code, a lot of the information was lost, because the local variable declarations inside the data-flow environment were disregarded. This originally caused the generated program to not be runnable.

Another problem was the use of referenced SCCharts. At the beginning I assumed that referencing an SCChart in data-flow worked the same way as it does in SCCharts without data-flow. This then entailed creating a node in data-flow, referencing an SCChart

inside this node, outputting the outputs from the referenced SCChart out of the node and then calling these outputs. This is of course very error-prone and did not work properly. However, for experienced SCCharts users this is the known way and should *theoretically* work yet does not. Using reference calls in data-flow the way it was intended yielded positive results.

Non-Fixed Problems

The first problem one encounters when using data-flow in SCCharts is that the visualization of larger models can take some time. While synthesizing the model works pretty fast, saving or loading a model causes the editor to slow down significantly for a few seconds as the visualization is loaded.

Furthermore is the visualization pretty overloaded. As can be seen in Figure 6.1 or in general in Section 5.2, the realization section of SCCharts, big models can become quite confusing. While nested SCCharts like in Figure 6.3 can help to create a more clear layout in bigger models, sometimes even smaller models are very cluttered and a nested SCChart might not clarify the model. In this case, manual or semi-manual adjustments might be of advantage.

More reasons for the cluttered look of the model are the partially redundant markings of relations. As Figure 6.4 shows, the model can have a lot of redundant markings that exist to keep the SCT file readable. Creating one big equation would help eliminate these markings but at the same time make corrections and adjustments nearly impossible. Therefore, I would suggest that these markings are removed from the visualization if the variable is merely a local variable in the data-flow. Similarly, one could not visualize every variable that is neither declared as an in- or output. Another idea would be to remove the marking if the variable is both an output and an input to actors. This however could remove some variables that the user would want to have displayed.

Another big problem is the use of hostcode for trigonometric functions. Not only should there be a possibility to calculate trigonometric functions without having to resort

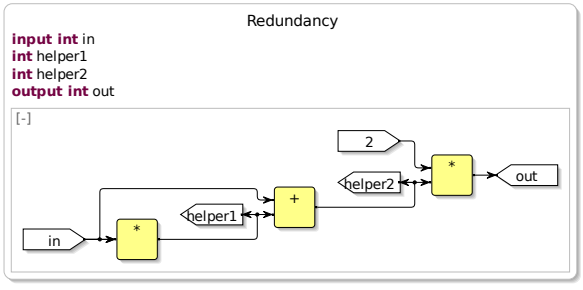
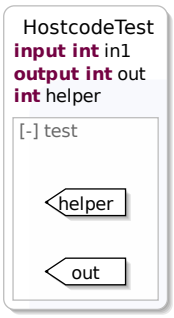


Figure 6.4. Redundant information in data-flow

```
1  scchart Hostcode {
2    input int in;
3    output int out;
4    int helper;
5
6    dataflow test:
7      helper = <sin(in)>;
8      out = 'sin(helper)';
9  }
```

(a) Hostcode implementations in SCCharts data-flow



(b) Hostcode disappearing in data-flow

Figure 6.5. Hostcode in SCCharts data-flow

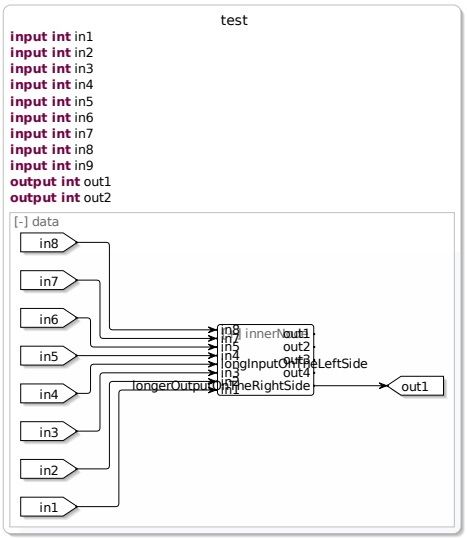


Figure 6.6. Cluttered inner node with too much text

to hostcode, currently hostcode is not even visualized in data-flow as Figure 6.5 shows. This occurs because the synthesis disregards the contents of the hostcode calls and simply forwards the code until the end of the synthesis and therefore does not know if any variables of the model are being used in the hostcode.

Even more visualization issues occur when using nested or referenced nodes inside data-flow. When the modeler adds more input or output variables or uses very long names for the variables, the size of the node does not change while it is closed. This leads to very cluttered nodes as in Figure 6.6 and prevents the user from opening the node to look at the contents of it as there are too many labels overlapping to click the button to open the node.

Chapter 7

Conclusion

This chapter shortly summarizes the results of this thesis in Section 7.1. It concludes with a look into future work in SCCharts in Section 7.2.

7.1 Summary

The aim of this thesis was to create a simulation of a quadcopter as a demonstrator for data-flow in SCCharts and to be able to compare it with other established modeling tools. With this demonstrator it was possible to evaluate the usability of SCCharts in the context of simulations and data-flow. This simulation managed to reveal flaws and errors in the data-flow environment such as the unclear layout of larger models and the lack of core elements of physical models like trigonometric functions as well as to evaluate its possible uses.

Data-flow in SCCharts might be a possible, time saving alternative to Ptolemy or Matlab in the future since creating a model is as easy as typing the equations in and watching the model visualize itself. While heterogeneous models might still be problematic for SCCharts, data-flow alone seems to be working pretty well and fluid, as long as the models don't become excessively large. Large models might be too unclear and fixing this is the most important issue. Since the modeler has almost no hand in the layout of the final product, it is difficult to give the model a reasonable layout from the creators point of view.

The simulation however should be developed further and tested more rigorously before actually being used to verify a flight controller of a quadcopter.

7.2 Future Work

As Chapter 6 has revealed, there are a lot of problems in data-flow in SCCharts as well as the simulation that was created for this thesis. Therefore, this thesis proposes some future projects to close this thesis.

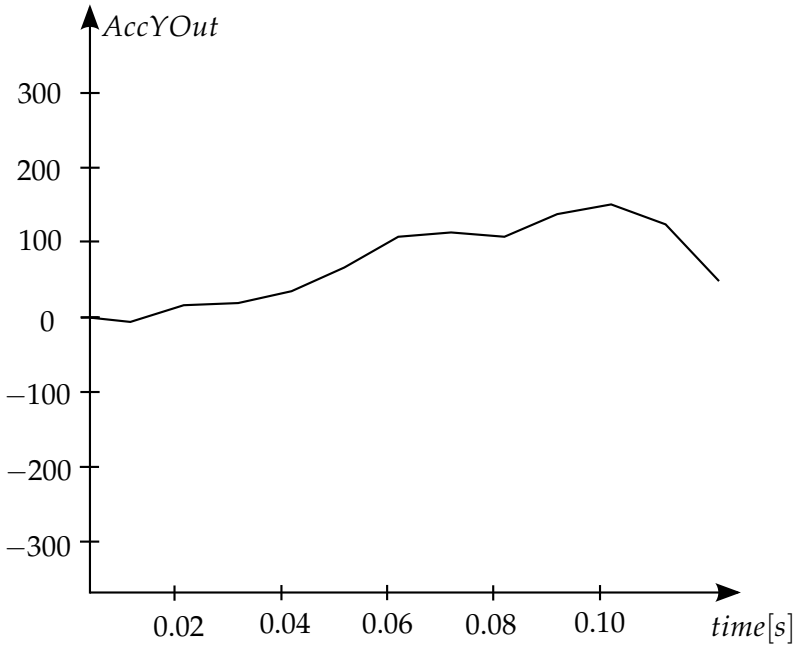


Figure 7.1. An example diagram displaying the acceleration in the direction of the y-axis of the quadcopter

7.2.1 Future Work on the Simulation

As a first step for the simulation it should be tested and overworked. While the first step to a good simulation would be to map the motor values that are put out by the Arduino to voltages that are then read by the motors, the distance calculation should be reworked as well as the current system is just a simplification.

To get the simulation to work with the quadcopter that was built for the project, a lot of physical properties would have to be determined such as the movement inertia of the quadcopter in every direction, the rotational inertia around each axis and the inertia, lift constant and drag constant of the rotors.

The current visualization of the simulation utilizes only the *data tables* component. While this component is able to show the values of all data, this does not necessarily help with a simulation. In simulations a visualization via diagrams like in Figure 7.1 is very helpful. This could be done via the KIELER Environment Visualization (KEV) [Kna10]. This project provides an interface for visualization that loads JSON Strings and create SVG graphics. An integration into this simulation could be very helpful.

For this simulation in particular, a better visualization using a 3D picture of a quadcopter in a room could be a good idea. This visualization could depict the position of the quadcopter in the room as well as its current angles at the current time like in Figure 7.2.

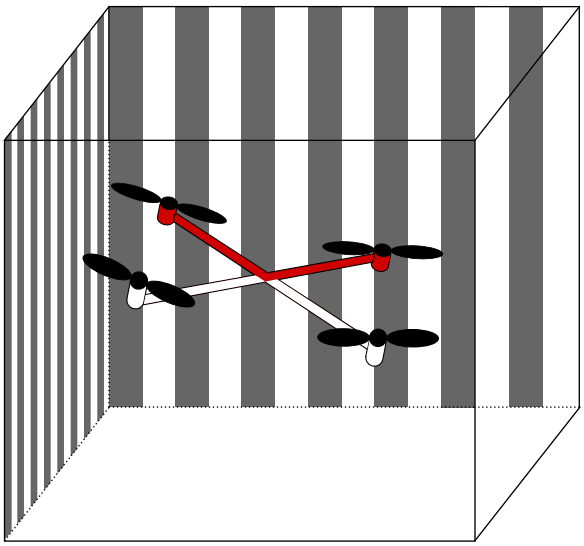


Figure 7.2. A three-dimensional representation of the quadcopter slightly tilted in a room

The simulation could further give information if the value of a variable has exceeded a certain limit, for example if the quadcopter has exited the boundaries of the room or is tilted more then desired. The creator should however analyze the costs and benefits of this work. As long as the simulation is not working properly, this visualization would not be very helpful.

7.2.2 Future Work on Data-Flow

Data-flow in SCCharts has a lot of room for expansion. Since the model with all the equations was already created beforehand, the model was very easy to implement in SCCharts. But, as mentioned in Section 6.3.2, there still exist some issues that need to be fixed.

The first step to making data-flow in SCCharts more usable is to improve the layout algorithm or to create a way to influence the layout more easily. Right now, unrelated variables can be located very close to each other while related ones are very far from each other away, which makes it hard to understand the model for outsiders. Even I as the modeler often times had problems understanding the flow of data in some models. Removing unnecessary markings as well as making it possible to group related variables without losing the connection to other variables would be a first step to better layouts. As an example, Ptolemy does an excellent job providing means to hide unnecessary information as well as to show connections via the ports of composite models. Data-flow in SCCharts already does the latter when using nested SCCharts but still struggles with mid-sized models. These models often struggle with unnecessary markings. Removing a marking

when it is not declared as an input or output variable of the diagram might be a good idea. Ptolemy handles this fairly well by only portraying the names of ports when the user hovers over this port. This or hovering over the connection to reveal the name instead of just the port could be implemented in SCCharts.

Furthermore, trigonometric functions and ways to integrate or derivate are often times very important for physical simulations, which is an important use case for data-flow. Therefore these functions should be more easily accessible. Since trigonometric functions are currently merely implemented via the use of hostcode, which in itself is quite problematic as the modeler might want a platform independent model, and integration and derivation are not implemented at all, SCCharts clearly lacks a lot of appeal. Implementing these kinds of functionalities though should not be hard to do. Furthermore, hostcode is not visualized in data-flow currently. Perhaps creating a small box signifying hostcode in the diagram could fix this. The hostcode could be written inside the box so the modeler knows what it does instead of it being a black box.

Lastly, Data-flow in SCCharts requires more demonstrators. At this point in time, data-flow in SCCharts has not been used by many people. A larger test – maybe as a part of a course – would be required to verify data-flow and to search for more flaws in the implementation or the visualization. An important part of this would be to let the students create both smaller and larger models and to conduct a survey afterwards.

In conclusion, this thesis describes with the help of a simulation for a quadcopter a demonstrator for data-flow in SCCharts. With this demonstrator the thesis compares Ptolemy and SCCharts as modeling and simulation tools. It shows the strength and weaknesses of SCCharts in the context of physical simulations.

Bibliography

[And15] Lewe Andersen. "'Quadrocopter Flight-Control Design using SCCharts'". submitted. Bachelor Thesis. Kiel University, Sept. 2015.

[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.

[BB91] Albert Benveniste and Gerard Berry. "The synchronous approach to reactive and real-time systems". In: *Proceedings of the IEEE*. 1991, pp. 1270–1282.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. "The Synchronous Languages Twelve Years Later". In: *Proc. IEEE, Special Issue on Embedded Systems*. Vol. 91. Piscataway, NJ, USA: IEEE, Jan. 2003, pp. 64–83.

[CLD05] Pedro Castillo, Rogelio Lozano, and Alejandro Dzul. "Stabilization of a mini rotorcraft with four rotors". In: *IEEE Control Systems Magazine* 25.6 (2005), pp. 45–55.

[CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. "A conservative extension of synchronous data-flow with state machines". In: *Proceedings of the 5th ACM International Conference on Embedded Software*. EMSOFT '05. Jersey City, NJ, USA: ACM, 2005, pp. 173–182. ISBN: 1-59593-091-4. DOI: 10.1145/1086228.1086261. URL: http://doi.acm.org/10.1145/1086228.1086261.

[Gol80] Herbert Goldstein. Classical Mechanics. Addison-Wesley, 1980.

[Hah02] Hubert Hahn. Rigid Body Dynamics of Mechanisms: 1 Theoretical Basics. Springer Science & Business Media, 2002.

[Har87] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274.

[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.

[HMA+13] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O'Brien. "Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation". In: *Proc. Design, Automation and Test in Europe Conference (DATE'13)*. Grenoble, France: IEEE, Mar. 2013, pp. 581–586.

[Hög14] Matthias Höger. "'Aspekte der mathematischen Modellierung eines Quadcopters'". http://num.math.uni-bayreuth.de/de/thesis/2014/Hoeger_Matthias/BA_Matthias_Hoeger.pdf. bachelor Thesis. University Bayreuth, Sept. 2014.

[Kna10] Stephan Knauer. "'KEV – KIELER Environment Visualization – Beschreibung einer Zuordnung von Simulationsdaten und SVG-Graphiken'". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/skn-st.pdf. Student research project. Kiel University, Department of Computer Science, July 2010.

[Lee03] Edward A. Lee. Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M03/25. University of California, Berkeley, CA, 94720, USA, July 2003.

[Lee06] Edward A. Lee. "The problem with threads". In: *IEEE Computer* 39.5 (2006), pp. 33–42.

[Luu11] Teppo Luukkonen. "'Modelling and control of quadcopter'". http://sal.aalto.fi/publications/pdf-files/eluu11_public.pdf. Bachelor Thesis. Aalto University, Aug. 2011.

[Mac15] Felix Machaczek. "'Collision Avoidance of Autonomous Safety-Critical Real-Time Systems'". submitted. Bachelor Thesis. Kiel University, Sept. 2015.

[Mot07] Christian Motika. "'Modellbasierte Umgebungssimulation für verteilte Echtzeitsysteme mit flexiblem Schnittstellenkonzept'". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-st.pdf. Student research project. Kiel University, Department of Computer Science, Oct. 2007.

[Mot09] Christian Motika. "'Semantics and Execution of Domain Specific Models—KlePto and an Execution Framework'". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, Dec. 2009.

[Pto14] Claudius Ptolemaeus, ed. System Design, Modeling, and Simulation using Ptolemy II. http://ptolemy.org/books/Systems. Ptolemy.org, 2014.

[SG78] G. E. Shilov and B. L. Gurevich. Integral, Measure, and Derivative: A Unified Approach. Trans. by Richard A. Silverman. Dover Publications, 1978.

[SSH14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. "Drawing layered graphs with port constraints". In: *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout* 25.2 (2014), pp. 89–106. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2013.11.005.

[TF96] George B Jr. Thomas and Roos L. Finney. Calculus and Analytic Geometry (9th ed.) Addison-Wesley, 1996.

[TM04] A Tayebi and S McGilvray. "Attitude stabilization of a four-rotor aerial robot". In: *Decision and Control, 2004. CDC. 43rd IEEE Conference on*. Vol. 2. IEEE. 2004, pp. 1216–1221.

[Uml15] Axel Umland. "'Konzept zur Erweiterung von SCCharts um Datenfluss'". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aum-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, Mar. 2015.

Instructions for the Simulation

.1 Simulating without the Arduino

To start the simulation without the Arduino you will require the following:

- ▻ Eclipse including a workspace with the repository of the semantics branch of the KIELER group
- ▻ The Eclipse Project for the Ptolemy data component located in the repository under Praktika/15ss-realtime/Ptolemy-Simulation/de.cau.cs.kieler.quadcopter.simulation imported into the Eclipse workspace
- ▻ The Ptolemy model located in the repository under Praktika/15ss-realtime/Ptolemy-Simulation/QuadcopterSimulation.xml **or**
- ▻ The SCCharts model located in the repository under Praktika/15ss-realtime/SCCharts Simulation
- ▻ The flight controller in SCCharts

Start the Eclipse instance and open the *SCCharts Simulation* perspective including the *Execution Manager*. Add a *Simple Ptolemy Simulation* and an *SCCharts/SCG Simulator (C)* to the execution and open the flight controller in the project explorer. Starting the execution should now execute the simulation. Alternatively, you can download the Quadcopter.execution file from the 15ss-realtime/SCCharts Simulation and execute this simulation file. If the input and output variables of the flight controller are not the same as these of the model, simply rename these variables in the SCChart.

If, instead of the Ptolemy model, you want to simulate the SCCharts model, you will merely need the SCCharts/SCG Simulator (C). Then open the combined flight controler/-model SCCharts file in the repository under Praktika/15ss-realtime/SCCharts Simulation.

.2 Simulating with the Arduino

For this simulation you will require the following:

- ▻ The Arduino with the QuadcopterController project installed and connected via USB to the simulating PC

- If you have Ubuntu installed: You will have to give some rights to the port the Arduino is connected to. To do that, go to the console and type *sudo chmod a+rw dev/[serialPort]* with [serialPort] being the port the Arduino is connected on. Typically this is ttyACM0 or ttyUSB0.
- Eclipse including a workspace with the repository of the semantics branch of the KIELER group
- The Eclipse Project for the Ptolemy data component and Arduino Communication data component located in the repository under Praktika/15ss-realtime/Ptolemy-Simulation/de.cau.cs.kieler.quadcopter.simulation imported into the Eclipse workspace
- The Ptolemy model located in the repository under Praktika/15ss-realtime/Ptolemy-Simulation/QuadcopterSimulation.xml **or**
- The SCCharts model located in the repository under Praktika/15ss-realtime/SCCharts Simulation

For this you will have to connect the Arduino with your PC first and install the QuadcopterController project on the Arduino. Otherwise, the method is the same as the above. However, instead of adding an SCCharts/SCG Simulator (C) data component, you will have to add the *Arduino Communication* data component. Then either add the Simple Ptolemy Simulation data component or the SCCharts/SCG Simulator (C) and open the SCCharts model (standalone! Not the combined version).

.3 Changing the Models

Changing the SCCharts model is quite easy with a working installation of SCCharts, preferably the nightly build since the current release v0.10.0 does not yet support dataflow.

Changing the Ptolemy model requires Ptolemy II 8.0.1 or newer. Furthermore, the *KielerIO* and *KielerIOFloat* actors have to be installed. These can be downloaded in the repository under Praktika/15ss-realtime/Ptolemy-Simulation and put into the new folder ptII[version]/ptolemy/actor/kiel. After saving the model, the XML file has to be put into the data component of the simulation. Copy the model XML file to the folder de.cau.cs.kieler.quadcopter.simulation/model. The model has to be named QuadcopterSimulation.xml to work. Furthermore, if you have installed version 0.10.0 or newer of Ptolemy, you have to search and replace every mention of *gui.MonitorValue* with *MonitorValue*.