INSTITUT FÜR INFORMATIK

Using SCCharts Models in Simulink to Model an Electronic Control Unit

Monty Santarossa, Steven Smyth,
Alexander Schulz-Rosengarten and
Reinhard von Hanxleden

Bericht Nr. 1903
July 2019
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Department of Computer Science
Kiel University
Olshausenstr. 40
24098 Kiel, Germany

Using SCCharts Models in Simulink to Model an Electronic Control Unit

Monty Santarossa, Steven Smyth,
Alexander Schulz-Rosengarten and Reinhard von Hanxleden

Report No. 1903
July 2019
ISSN 2192-6247

E-mail: {stu121264,ssm,als,rvh}@informatik.uni-kiel.de

Technical Report

Abstract

When constructing an electrical racing car, special attention needs to be directed to the development of its engine control unit. Functionality of the motor-torque calculation and the integration of advanced driver assistance systems are crucial for the speed handling and hence, the safety of the car. The Kieler Formula Student Team *Raceyard*, which since 2011 has been constructing electrical racing cars annually, so far designed and tested its controller model in the popular commercial modeling software MATLAB/Simulink.

This work shows how a functionally equivalent system can be designed by utilizing the visual synchronous language SCCharts in the academic open-source project KIELER. A complete controller model is modeled in KIELER and validated to behave the same as the original controller both in Simulink directly as well as in the 3D simulation environment IPG Carmaker. Tests on the performance of both controllers show that while a slowdown can be observed when comparing the generated C Code, simulation time in IPG Carmaker only increases by a negligible factor.

KIELER's developing and testing capabilities for synchronous models can therefore be considered a valuable tool in the process of designing, tuning and documenting such a controller model.

Contents

1 Introduction

The Formula Student is an international design competition for students, where every year the aim for each team is to design, construct and test a race car. These race cars are then compared and judged during official events. Points are awarded for static disciplines, such as preparing a business plan and a cost report, and for dynamic disciplines, where the car has to race on different courses. Since 2005 Team *Raceyard* takes part in the Formula Student as the official team of the University of Applied Sciences Kiel. Beginning with the season 2011/2012 *Raceyard* has exclusively designed electric cars - a trend that is continued in the current season with the additional challenge of using a four-wheel drive for the first time in the team's history.

Designing an electric car means that each of the motors has to be controlled by a program instead of by mechanical devices. The signals from the various sensor boards in the car as well as the suggestions from the driver, i.e. acceleration pedal position, brake pedal position and if necessary the steering angle, have to be converted into four torques for the four electric motors. Further aspects to be considered are power limitations dictated by the Formula Student rulebook, the traction control system, torque vectoring that allows a sophisticated torque distribution to avoid over- and understeer and a system for recuperative braking, where the motors act as a generator to charge the battery. The program that accomplishes all of this as well as handling communications between the boards inside the racing car is by *Raceyard* internally referred to as the Electronic Control Unit (ECU).

For the task of designing and testing the control systems of the ECU, *Raceyard* uses the MathWorks' modeling software Simulink. Integrated into MATLAB, Simulink provides a block diagram environment for modeling and testing dynamic systems. Of special importance for *Raceyard* are the built-in blocks for PID controllers and the provision of scopes and displays to view every signal inside the system during and after runtime. Furthermore, those signals can be saved externally into MATLAB-arrays for later analyses. Verifying as well as fine-tuning the ECU is done via the simulation software IPG Carmaker. IPG Carmaker provides a virtual 3D environment and simulated drivers of varying abilities so that the effects of the ECU can intuitively be seen on the 3D car model. Due to integration of IPG Carmaker into Simulink, values from the simulation such as the velocity of the car or the forces acting upon it can be measured and saved using the previously described scope- and display-blocks in Simulink. This practically allows for storing entire test runs. Following the test-phase, the Simulink model is then converted into C code and put on an STM32F40 micro controller inside the car.

SCCharts is a synchronous language that was designed for safety-critical applications [11]. It can be seen as a successor to SyncCharts [1], which itself is a graphical statecharts variant of Esterel [2], combining the semantics of Esterel [3] with the graphical

notation of Harel's statecharts [5]. The most complete implementation of the SCCharts language is included in the academic, open-source project Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1] as the KIELER SCCharts Editor. The editor includes all tools a developer needs to develop, simulate, and compile SCCharts. It also supports different code generation approaches and pre-defined compilation chains for various target platforms.

The goal of this work is to evaluate whether the KIELER modeling environment can be used to design an ECU-model equivalent to those already used by *Raceyard*. The performance of the KIELER model is compared against the existing models, both when integrating the KIELER model as a System in Simulink as well as when generating C code for the micro controller from Simulink.

To this end the following workflow was applied: The subsystems of the *Raceyard* controller were remodeled as SCCharts using KIELER. The KIELER SCCharts were then converted into C Code and integrated into Simulink as several subsystems. These steps are presented in Chap. 2. The model was validated by testing its generated C Code as well as comparing the results in Simulink and IPG Carmaker to the original controller model in Chap. 3. The performance of both models as well as the performance of their resulting codes were measured in different environments in Chap. 4. Related work is introduced in Chap. 5.

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

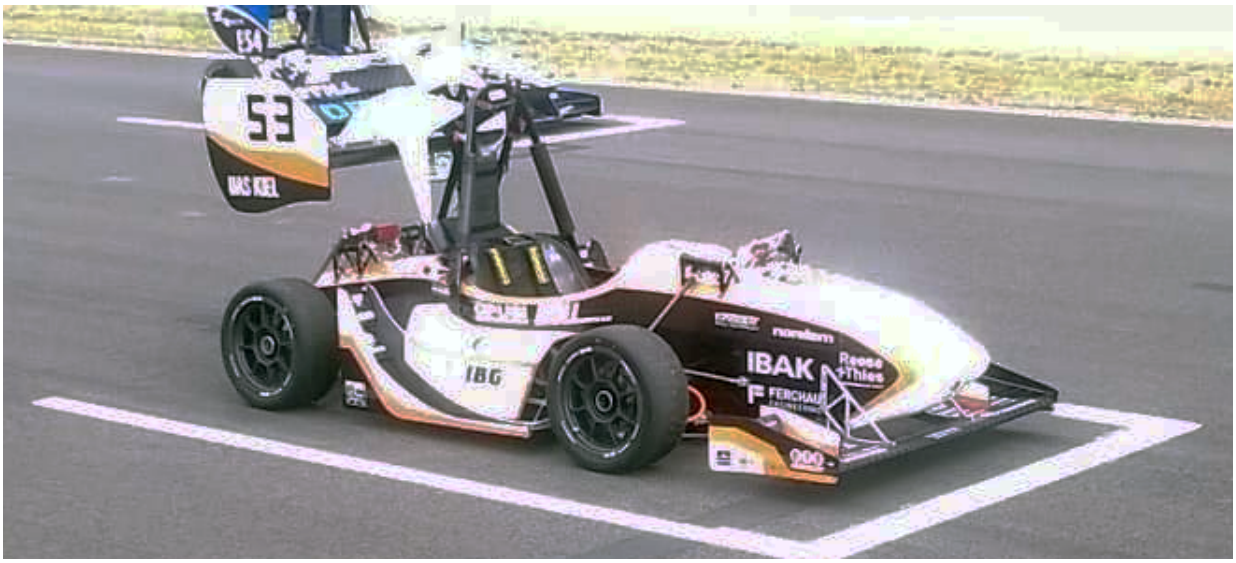


Figure 1.1: The T-Kieler18 B racing car built by Team *Raceyard* in the Season 2018/2019

2 The Control System Model

This chapter introduces the Control Systems model as used by *Raceyard*. The subsystems and the resulting C code for the ECU are explained in Sec. 2.1 and Sec. 2.2, respectively. In Sec. 2.3 the subsystems are remodeled as SCCharts in KIELER. Sec. 2.4 looks at the C Code generated from the SCCharts and Sec. 2.5 shows how it is integrated into Simulink. A different approach of modeling the SCCharts as dataflow is introduced in Sec. 2.6.

2.1 The Control Systems Model in Simulink

In its most simple form, the controller is considering only the positions of the brake and the acceleration pedal from the driver input. If the brake pedal is pressed to any greater amount than 5%, no (positive) output torque is given to the motors. In any other case the output torque is scaled proportionally to the position of the acceleration pedal and evenly distributed to the four electric motors. The maximal output torque is then limited only by the maximal torques of the motors (28 Nm for each one) and the power limitations given by the Formula Student rulebook (80 kW at any given moment).

Note that the described controller does not include a traction control system, torque vectoring or recuperative braking. In fact, this controller model does not even include any sort of P, PI or PID controller. However, while being simplistic, this is the controller model that *Raceyard* used extensively for testing in the real car. Using a simple but

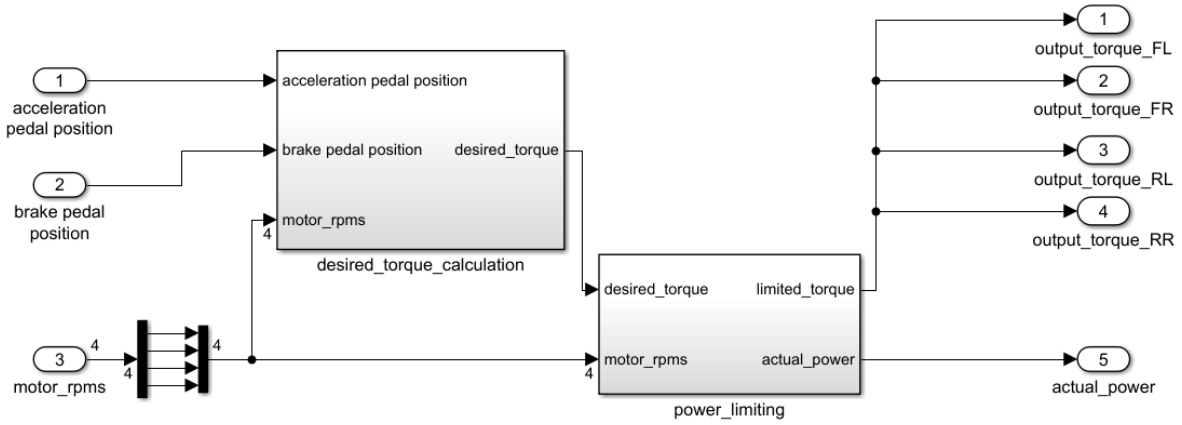


Figure 2.1: The top layer of the controller model in Simulink

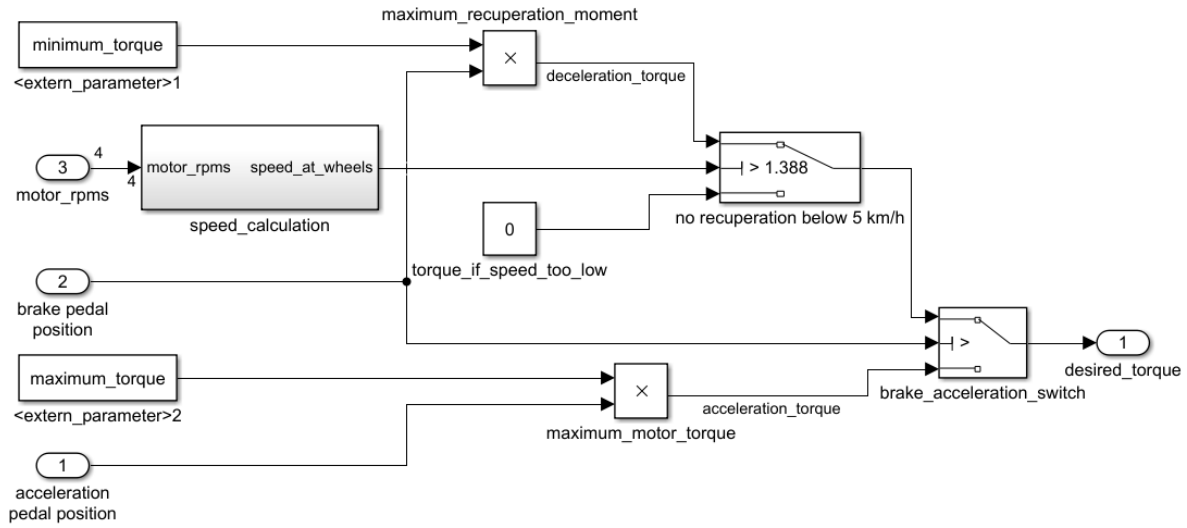


Figure 2.2: The subsystem desired torque calculation in Fig. 2.1

robust controller proved to be helpful when evaluating other, possibly more critical compartments of the car.

Fig. 2.1 shows the Simulink model of the described controller. The model needs three inputs: Input one and two are the positions of the acceleration and the brake pedal respectively, coded in a range from 0 to 1; 1 meaning the pedal is pressed down completely. The third input is a 1x4 vector consisting of the current revolutions per minute of each of the four electric motors. The model itself is divided into two subsystems. The three inputs are converted into a desired output torque in the first subsystem desired torque calculation. This torque is a single value equal for every motor. It is then checked in the second subsystem power limiting whether the desired torque would lead to a needed mechanical power that is greater than a given power limit. If this is the case, the output torque is limited; if it is not the case, the output limited torque of the subsystem equals its input desired torque. This single output value is then propagated to the four outputs, resulting in four equal torques for the motors. Furthermore, the subsystem power limiting calculates the needed power for the limited torques. The calculated power value is then given first to the output actual power of the subsystem and then to the output with the same name of the complete model. Note that this output is only a control value for testing and is not influencing the behavior of the car in any way.

2.1.1 Desired Torque Calculation and Speed Calculation

Fig. 2.2 shows the desired torque calculation subsystem of the Simulink Controller. In essence, the subsystem checks whether the driver wants to brake or accelerate and sets the desired output torque accordingly. The decision of braking or accelerating is done via the brake acceleration switch at the end of the subsystem. A brake pedal position of

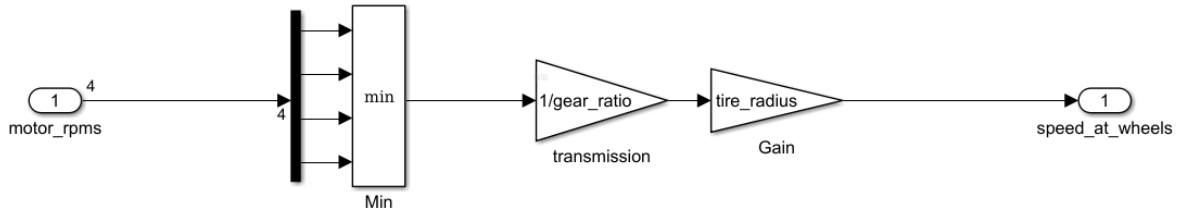


Figure 2.3: The subsystem speed calculation in Fig. 2.2

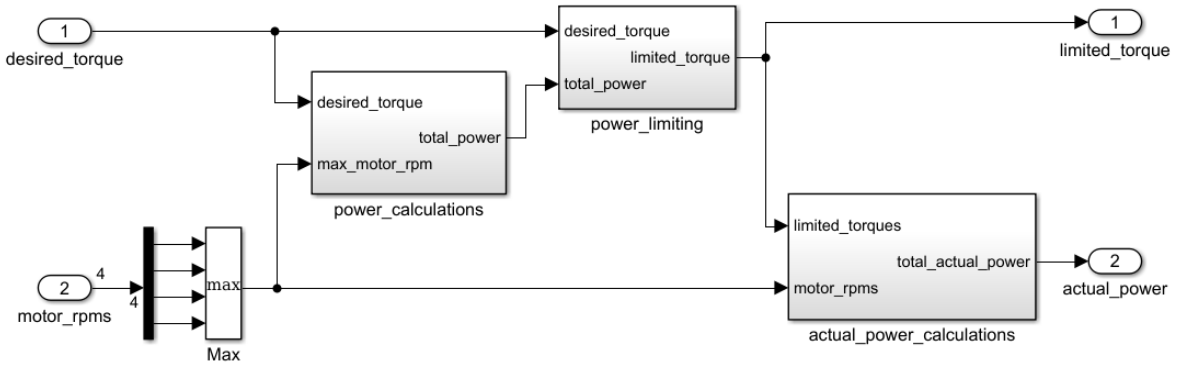


Figure 2.4: The subsystem power limiting in Fig. 2.1

more than 5% is interpreted as the intention to brake.

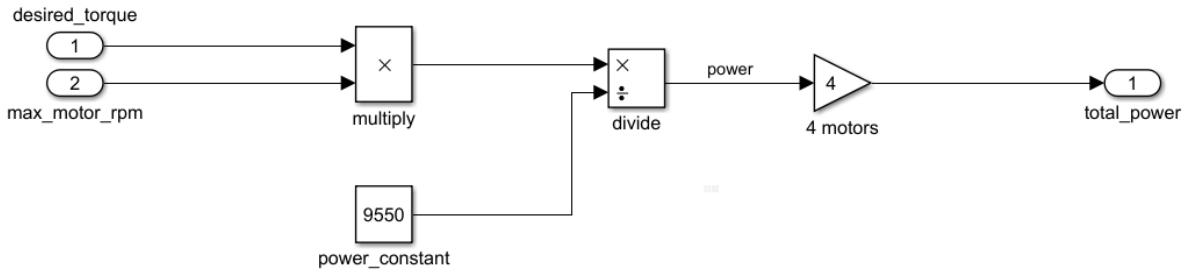
While, as stated in the beginning of this chapter, recuperative braking is not supported by the controller, a basic test architecture for it already exists and is described by the top part of the subsystem in Fig. 2.2. At first, the rpm of the slowest motor is detected and converted to the longitudinal velocity in m/s of the corresponding wheel. The formula for this is:

$$velocity = \frac{rpm \cdot tire\_radius}{gear\_ratio}$$

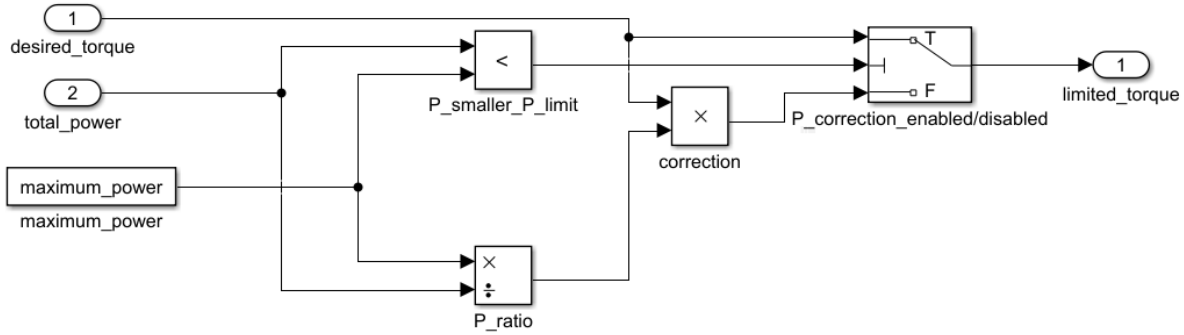
The Simulink model for the calculation is shown in Fig. 2.3. Note that this way of calculation the velocity does not hold up in the real world because of tire slip. If in simulated circumstances the calculated velocity is above the threshold for recuperative braking, e.g. $5\,km/h \approx 1.388\,m/s$ as stated in the Formula Student Rulebook, the amount of negative torque for recuperation is proportional to the position of the brake pedal. The same principle applies when the driver wishes to accelerate. This is described by the bottom part of the subsystem. The acceleration pedal position is multiplied by the maximal torque available for the motors (28 Nm in our case), so that a pedal position of 100% leads to the maximal possible output torque.

2.1.2 Power Calculation and Power Limiting

After the desired torque is calculated in the subsystem desired torque calculation, the adequately named subsystem power limiting, shown in Fig. 2.4, limits the output torque,



(a) The subsystem power calculation



(b) The subsystem power limiting

Figure 2.5: The subsystems of Fig. 2.4

if the needed mechanical power needed for the desired torque would exceed a given limit. To that end, the subsystems power limiting itself is split into three subsystems: power calculation, power limiting and actual power calculation. It should be noted that only the first two subsystems influence the behavior of the car, since actual power calculation only affects the testing value actual power.

The subsystem power calculation is shown in Fig. 2.5a and models the formula:

$$power = \frac{torque \cdot rpm}{9550}$$

Here $\frac{1}{9550}$ is the constant for motor torques when converting from Nm/min to kW. In order to ensure an over-approximation, this calculation is done using the maximum of the four motor rpms. The result is then multiplied by four to get an approximation of the total needed mechanical power for the desired output torque. It should be noted that due to friction etc. the actual electrical power drawn from the accumulator is higher than the used mechanical power. The ratio between the two is described as energy conversion efficiency and is constantly changing depending on multiple conditions. In the simple *Raceyard* controller this ratio can be accounted for by choosing the parameter maximum_power accordingly.

Should the over-approximated mechanical power exceed the given maximum_power, the output torques are limited accordingly in the subsystem power limiting as seen in

Fig. 2.5b. The ratio between the needed and the allowed power is calculated, after which the output torque is multiplied with this ratio.

The subsystem actual power calculation works in exactly the same way as power calculation, but instead of the desired torque gets the limited torque as input. If limitation of the torques was necessary the output power of this subsystem should always be equal to the maximum power, data type imprecisions excepted.

2.2 The Simulink Model in C

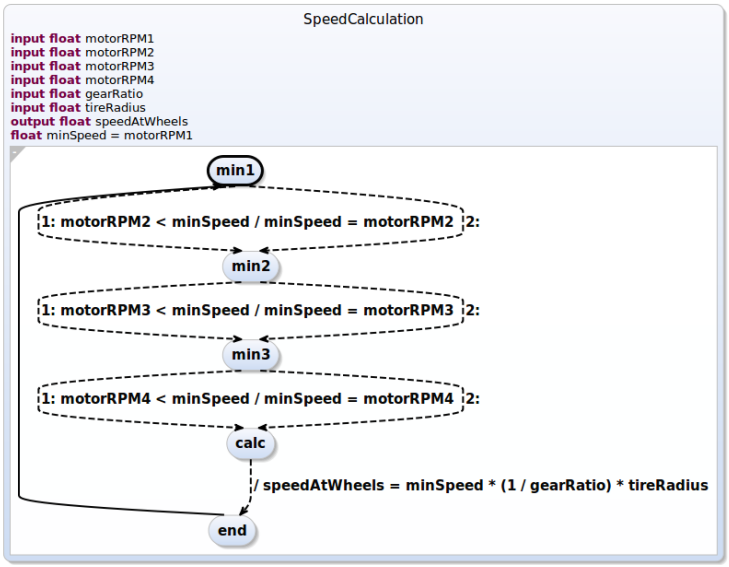
In order for the Simulink model to run on the micro controller, it had to be converted into C code, which is then integrated into the overall ECU program. Simulink provides tools to automatically generate C code. For the purposes of Raceyard, this results in three files: a typedef file, that is not further discussed here, one header and one source file. The header file contains declarations of the initialize()-function and the step()-function as well as definitions of the input struct and the output struct of the model.

The initialize()-function as defined in the source file is needed if the model uses blocks with designated starting conditions. The step()-function as defined in the source file contains the logic of the model. The input and output struct have as their respective members the input and output signals of the Simulink model as seen in Fig. 2.1. External parameters such as maximum power or gear_ratio (refer to Fig. 2.5b and Fig. 2.1 respectively) are declared as global variables in the header file and initialized with their standard values in the source file.

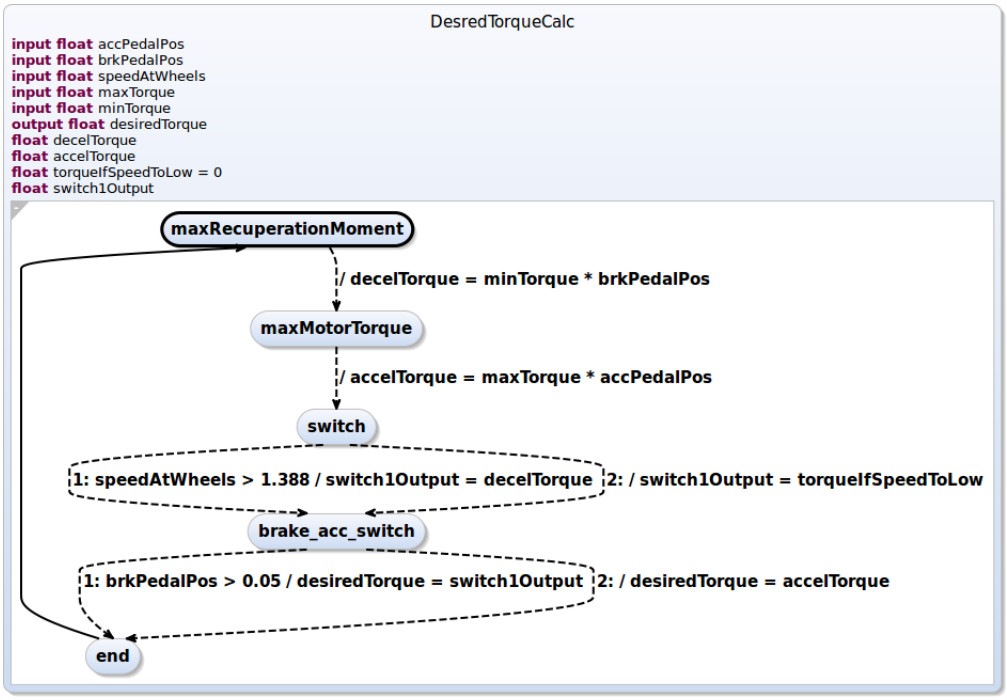
One run of the model can then be realized in the following way: The initialize()-function is called once as soon as the ECU starts. The variables of the input struct are set according to the current values provided by the real sensors or the simulation. Next, the step()-function is called, which results in the controller running the model and writing the calculated outputs into the output struct. From there, the outputs can be read and sent to the motors by the ECU. The process of writing into the input struct and calling the step()-function is repeated for as long as the ECU runs.

2.3 The Control Systems Model in KIELER

Given that the controller model consists of five subsystems, the decision was made to separately remodel those five subsystems rather than the whole controller itself. Since the resulting models are practically black boxes in Simulink (refer to Sec. 2.5 and Chap. 6), this approach not only guaranteed a faster success in debugging, but also closely resembled a possible future design process, where most likely only certain elements of the controller are modeled and tested in SCCharts. It should also be noted that in the following SCCharts models, readability and resemblance of the original Simulink models were given priority. Most of the subsystems describe one prolonged mathematical function that could be realized with only a single state.

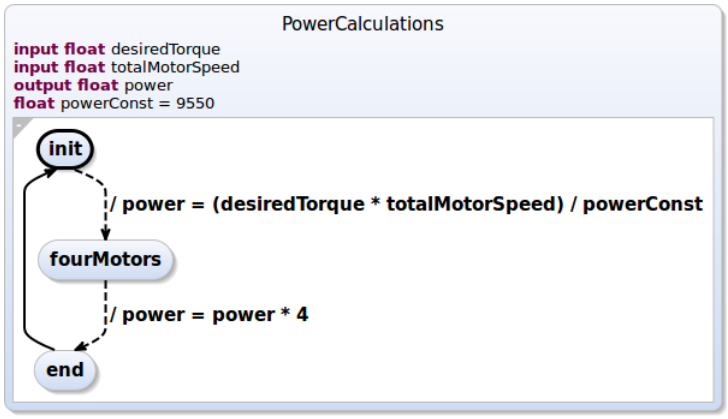


(a) The subsystem speed_calculation remodeled in SCCharts (compare to Fig. 2.2)

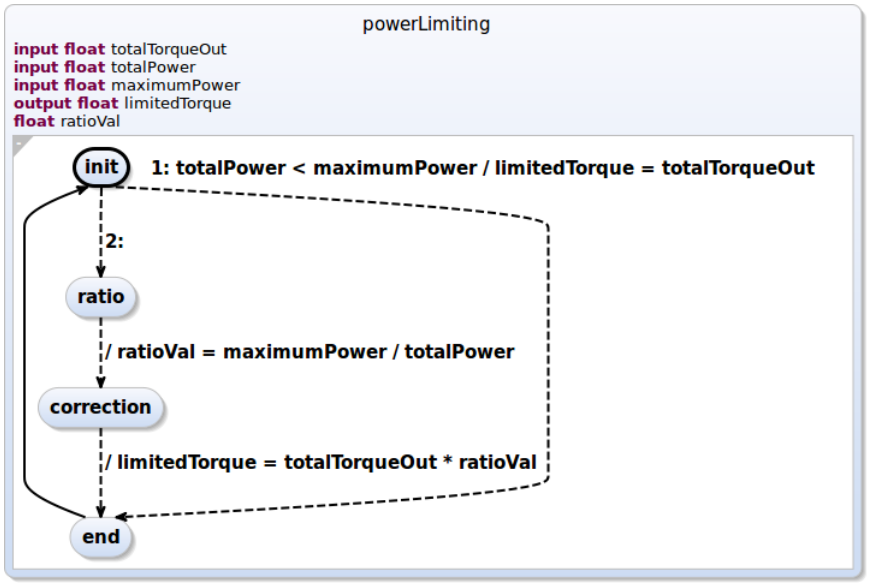


(b) The subsystem desired_torque_calculations remodeled in SCCharts (compare to Fig. 2.3)

Figure 2.6: The subsystem desired_torque_calculations in SCCharts



(a) The subsystem power_caluclations remodeled in SCCharts (compare to Fig. 2.5a)



(b) The subsystem power_limiting remodeled in SCCharts (compare to Fig. 2.5b)

Figure 2.7: The subsystem power_limiting in SCCharts

Fig. 2.6a shows the SCCharts resembling the speed_calculation subsystem. The first three states model the functionality of the min-Block of the original model while the fourth combines both gain-Blocks. Fig. 2.6b shows the desired_torque_calculation subsystem that can be seen in Fig. 2.2. The first and second state calculate the deceleration_torque and acceleration_torque of the original Simulink model respectively. The third state serves as the recuperation-switch, while the fourth act as the brake-switch.

The power_calculation subsystem as shown in Fig. 2.7a is a straightforward implementation of the power formula multiplied by four. The actual_power_calculation SCCharts model is, input names excepted, identical to the power_calculations SCCharts model. Finally, the power_limiting subsystem as an SCCharts model is shown in Fig. 2.7b. The first, second and third state serve as the less-than-block, the divide-block and the multi-

ply-block of the original Simulink model respectively. The fourth state act the switch at the end of the original model.

2.4 The KIELER C Code

To illustrate the structure of the KIELER C code, the code generated from the desiredTorqueCalculation SCCharts model is used exemplary for all the SCCharts modeled in Sec. 2.3. By default the SCCharts C code generator creates one header file and one source file.

Listing 2.1: C tick data struct

```
1  typedef struct {
2    double accPedalPos;
3    double brkPedalPos;
4    double speedAtWheels;
5    double maxTorque;
6    double minTorque;
7    double desiredTorque;
8    double decelTorque;
9    double accelTorque;
10   double torqueIfSpeedToLow;
11   double switch1Output;
12   char _g1;
13   char _g2;
14   char _g3;
15   char _g4;
16   char _g6;
17   char _g8;
18   char _GO;
19   char _cg1;
20   char _cg3;
21   char _TERM;
22   char _pg4;
23 } TickData_DesredTorqueCalc;
24
25 static inline void reset_DesredTorqueCalc(TickData_DesredTorqueCalc* d);
26 static inline void logic_DesredTorqueCalc(TickData_DesredTorqueCalc* d);
27 static inline void tick_DesredTorqueCalc(TickData_DesredTorqueCalc* d);
```

The header file as seen in List. 2.1 defines the reset(), tick() and logic() functions. It also defines a TickData struct that contains all the inputs, output and local signals defined in the SCCharts model as well as additionally needed internal signals.

Listing 2.2: C tick and reset functions

```
1 #include "DesredTorqueCalc.h"
2
3 static inline void reset_DesredTorqueCalc(TickData_DesredTorqueCalc* d) {
4   d->_GO = 1;
```

```
5   d->_TERM = 0;
6   d->_pg4 = 0;
7 }
8
9 static inline void tick_DesredTorqueCalc(TickData_DesredTorqueCalc* d) {
10  logic_DesredTorqueCalc(d);
11  d->_pg4 = d->_g4;
12  d->_GO = 0;
13 }
```

The source file as seen partly in List. 2.2 defines the mentioned functions. As the name suggests, reset() is called in order to bring all signals into their initial state. The logic() function is called via the tick() function and executes the internal logic of the SCCharts model, i.e. updating all the signals.

Listing 2.3: C wrapper function

```
1  #include "DesredTorqueCalc.c"
2
3  static TickData_DesredTorqueCalc d_dtc;
4  static bool dtc_init = true;
5
6  static inline void wrapper_DesredTorqueCalc(double accPedalPos, double brkPedalPos, double
     speedAtWheels, double maxTorque, double minTorque, double *desiredTorque)
7  {
8
9    if (dtc_init) {
10     reset_DesredTorqueCalc(&d_dtc);
11     dtc_init = false;
12   }
13   d_dtc.accPedalPos = accPedalPos;
14   d_dtc.brkPedalPos = brkPedalPos;
15   d_dtc.speedAtWheels = speedAtWheels;
16   d_dtc.maxTorque = maxTorque;
17   d_dtc.minTorque = minTorque;
18   tick_DesredTorqueCalc(&d_dtc);
19
20   *desiredTorque = d_dtc.desiredTorque;
21 }
```

For the purposes of this work a third file was generated: The wrapper.c file as seen in List. 2.3 contains an interface to easily allow execution of one tick of the SCCharts logic by calling wrapper function. An instance of the according TickData is created and initialized by calling the reset() function. The input signals of the SCChart model are set according to the given parameters and the tick() function is called, before the outputs are returned via call by reference. Since the reset() function should be executed only once in the first tick, there is an additional global variable init that is initialized with 'true' and will be set to 'false' after reset() is called, ensuring that reset() is called only once.

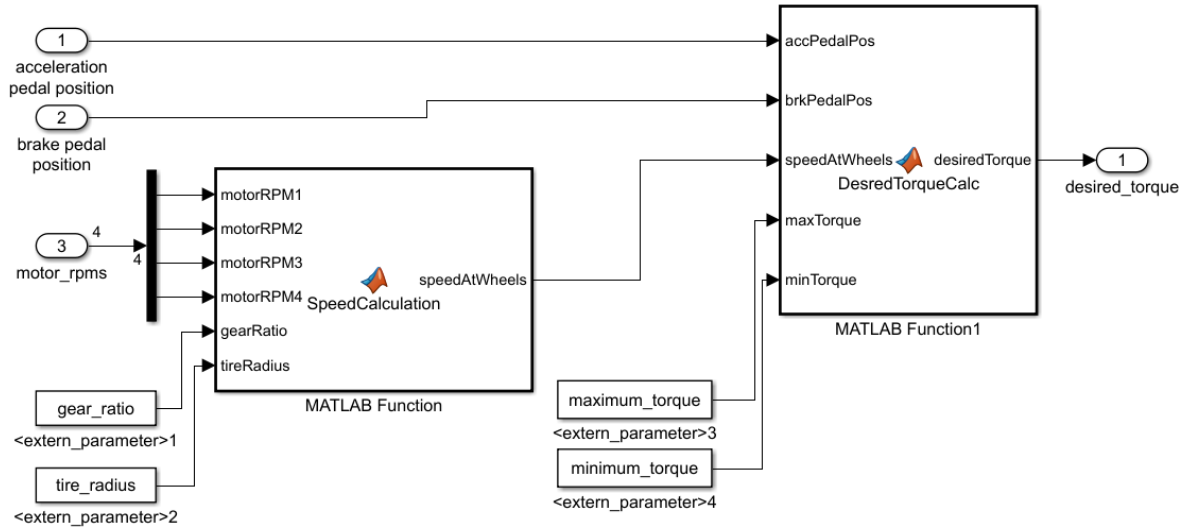


Figure 2.8: The subsystem desired torque calculation of the KIELER Controller

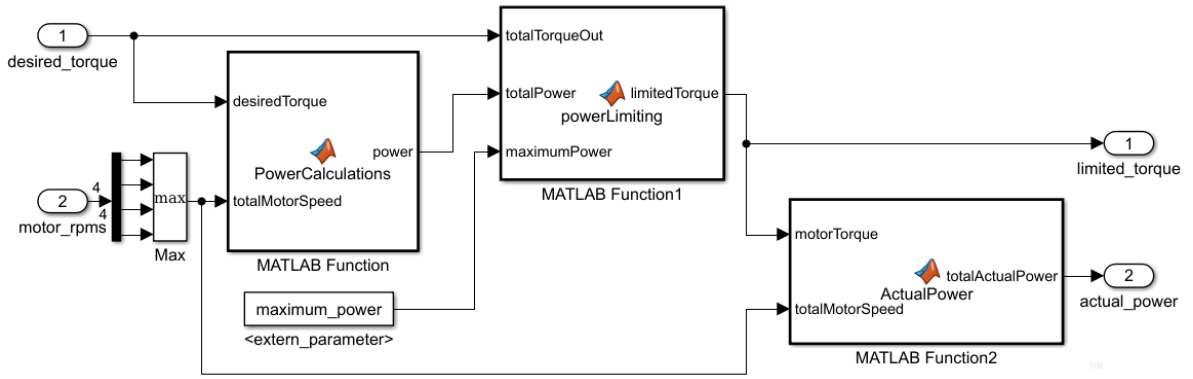
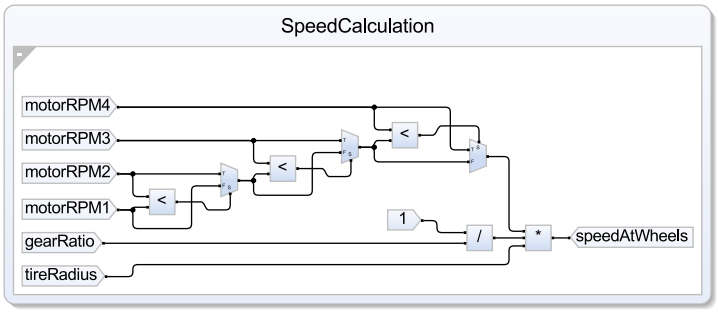


Figure 2.9: The subsystem power Limitation of the KIELER Controller

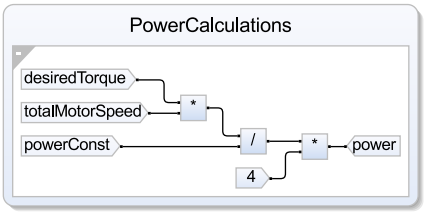
2.5 Integrating KIELER C Code into Simulink

MATLAB and by extension Simulink provides various ways of including external code. Though the possibility to model state dynamics through the Level-2 MATLAB S-Function or the S-Function seemed promising, the first executed too slowly in practice while the latter required more setup work than were deemed necessary for the purposes of this work. Therefore, the fast and more streamlined approach of utilizing a MATLAB function was chosen. In accordance with the instructions listed on the official Mathworks Documentations website[1], the wrapper C code can be integrated by using the MATLAB coder.cinclude function to include header files, the coder.updateBuildInfo with the parameter 'addSourceFiles' to include source files, the coder.ceval function to call

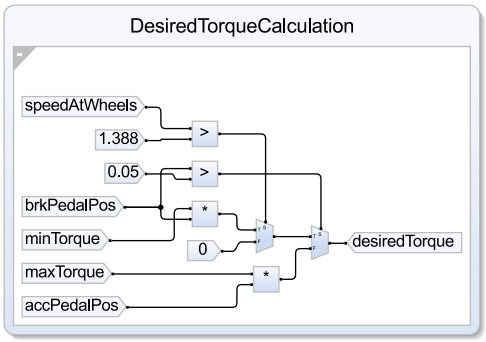
[1] https://de.mathworks.com/help/simulink/ug/incorporate-c-code-using-a-matlab-function-block.html



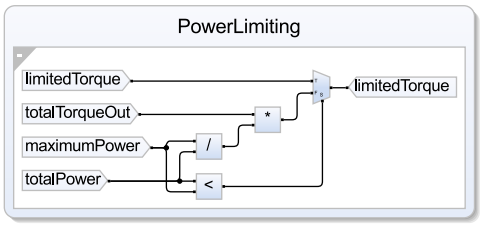
(a) SCCharts Dataflow model of the speed calculation from Fig. 2.6a



(b) Dataflow model of the power calculation from Fig. 2.5a



(c) LSCCharts dataflow model of the desired torque calculation from Fig. 2.6b



(d) Dataflow model of the power limiting from Fig. 2.5b

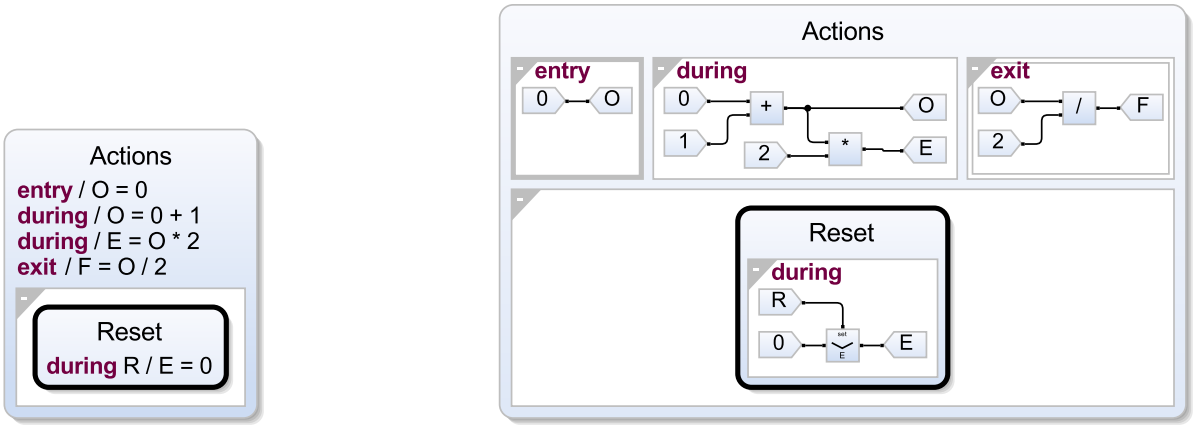
Figure 2.10: SCCharts dataflow models for critical model parts introduced in Sec. 2.3

outside functions and the coder.ceval function to realize call by reference.

MATLAB Functions can be integrated into the Simulink model via the MATLAB function block. Therefore, the KIELER controller model in Simulink as seen in Fig. 2.8 and Fig. 2.9 consists of five such blocks, two in the subsystem desired torque calculation and three in the subsystem power limiting. Evidently some of the MATLAB function blocks feature additional inputs that their original subsystem counterparts lack. As described in Sec. 2.3, these inputs exist to account for the global parameters such as maximum power that can be set from outside the Simulink model.

2.6 Dataflow in KIELER

Calculation-based models often only use instantaneous transitions to execute a sequence of assignments. This pure dataflow usage within a state-based language can sometimes be cumbersome to model and impair the overview. With recent SCCharts upgrades the modeler is now capable to create dataflow or hybrid models directly. This dataflow extension is a new extended feature of SCCharts. The feature will be transformed away during compilation using the standard compilation approaches available in KIELER



(a) Actions in the classical state-based SCCharts style

(b) View of the Fig. 2.11a model with actions displayed as dataflow. The operator used in Reset performs an update on E.

Figure 2.11: Different action views of the same SCCharts model

SCCharts [8, 11]. Basically, each dataflow assignment is translated into a during action, which will be translated as usual [7]. Hence, as long as the enclosing state (or statemachine) is active, a *dataflow region* is executed in every tick.

Fig. 2.10 shows four dataflow variants of the speed calculation (in Fig. 2.10a), the power calculation (in Fig. 2.10b), the desired torque calculation (in Fig. 2.10c), and the power limiting (in Fig. 2.10d) presented in Sec. 2.3. In each model you can see how the required data inputs on the left are combined to form the desired output on the right. SCCharts' new modeling features aim to encourage modelers to embrace a more hybrid-like modeling style, where state-based solutions are modeled with statemachines and calculation-based parts are modeled (or at least visualized) in dataflow. Both incarnations can co-exist.

However, while it might be preferable in some contexts to be able to model dataflow relations in dataflow regions, both worlds can also benefit from each other through dedicated views without modeling in the other world directly. For example, state-based actions from superstates can be displayed as dataflow, even if modeled in SCCharts' classical state-based sense. Fig. 2.11 shows different views on the same SCCharts model. In Fig. 2.11a the classical state-based syntax with textually displayed actions is depicted. The dataflow counter-part is shown in Fig. 2.11b, using the same underlying model. Only the graphical syntax changed.

Another example of this principle is the induced dataflow view presented elsewhere [12]. Here, dataflow relationships in pure state-based models are made visible explicitly during the view synthesis. Again, there is no need to modify the original state-based model.

3 Validation

In order to ensure that the KIELER controller works as intended, it has been tested against the *Raceyard* controller it modeled after. Ideally, the KIELER controller works such that no difference in the outputs can be observed when compared to the *Raceyard* controller. Comparisons were done directly in Simulink, where full control over the inputs is given, as well as in the 3D simulation software, where the controller can be tested under the complex conditions similar to those in a real car.

3.1 Validation in Simulink

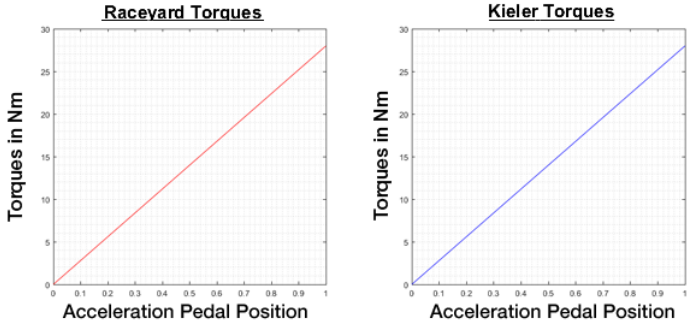
The goal for the KIELER controller is to function exactly as the *Raceyard* controller. Therefore a setup has been chosen where both controllers are assigned the same inputs and their outputs are compared against each other. Since there are three different kinds of inputs, acceleration pedal, brake pedal and motor rpms, the effects of all three have been tested.

If the brake pedal is not pressed and no power limiting has to be applied because of too high motor rpms, an acceleration pedal position of 0% should result in output torques of 0 Nm while a acceleration pedal position of 100% should result in the maximal 28 Nm possible for the motors with a linear interpolation in between. As can be seen in Fig. 3.1a, both the *Raceyard* and the KIELER controller accomplish this. As stated in Sec. 2.1, a brake pedal position of more than 5% is considered as the intention to brake. In this case a torque of 0 Nm should be given out. As Fig. 3.1b shows, this again is accomplished by both controllers.

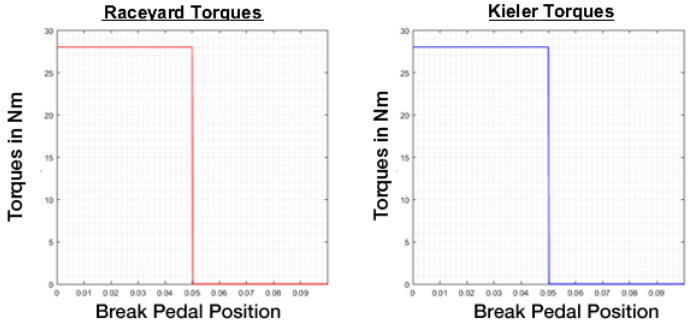
3.2 Validation in IPG Carmaker

For testing the power limitation, a theoretical scenario is considered, where the max motor rpm is linearly growing from 0 to 11000 and the mechanical power equals the electrical power drawn from the accumulator. According to the power formula presented in Sec. 2.1, if the acceleration pedal is pressed 100% (and therefore an output torque of 28 Nm for each motor is desired), power limitation should start as soon as the max motor rpm reaches

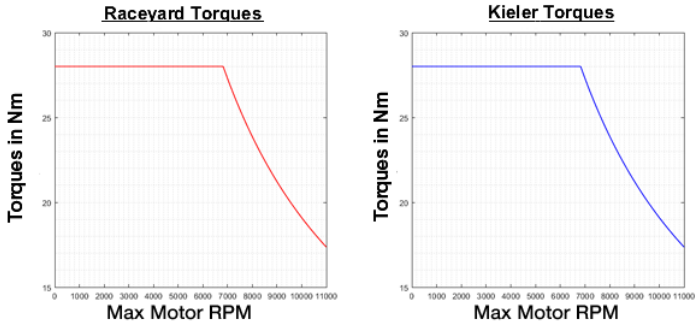
$$\frac{80kW \cdot 9550}{4 \cdot 28Nm} \approx 6821rpm.$$



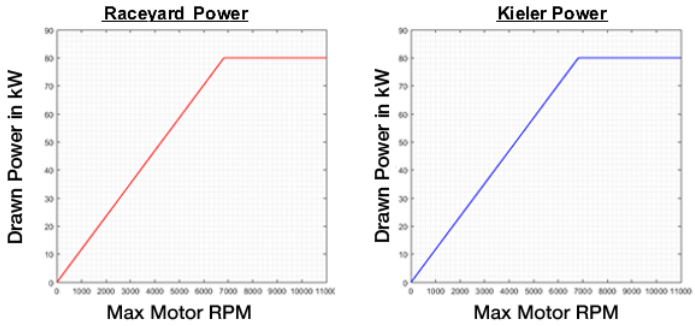
(a) Output Torques according to the acceleration pedal position



(b) Output Torques for full acceleration and different brake pedal positions



(c) Limiting Torques w.r.t. max motor rpm for full acceleration



(d) Calculated power drawn from the accumulator for full acceleration and different max motor rpms

Figure 3.1: Comparisons between the Raceyard and the KIELER Controller in Simulink

Figure 3.2: One of the validation tracks made in IPG Carmaker

As Fig. 3.1c shows, this holds true for both controllers. Fig. 3.1d further shows that the needed mechanical powers which both controllers calculate, stays at 80 kW after limiting is necessary. This means the torques are limited in such a way that the maximal possible target torque (within the limitations of the calculation) is given to the motors. In the end, where the max motor rpm reaches 11000 rpm, this is

$$\frac{80kW \cdot 9550}{4 \cdot 11000rpm} \approx 17.36Nm.$$

To compare both controllers under conditions that are similar to those in a real car, the simulation software IPG Carmaker was utilized. Since IPG Carmaker features a 3D rendering environment, the equivalence of both controllers for a certain test run can be established by comparing the paths driven: Given the same car, the same track and the same virtual driver, both vehicles should behave exactly the same.

This test was done for several tracks, one being pictured in Fig. 3.2. One run on that track consisted of the racing car driving for 200 meters. As can be seen in Fig. 3.3, both controllers result in the same steering angle as well as the same velocity at each part of the track and they both needed 16.8 seconds to complete the 200 meters. Those values stand exemplary for the exact same behavior of both controllers that could be established during all the test runs. This equivalence could also be seen directly in the 3D environment. IPG Carmaker provides a feature, where the path a car took during a certain test run can be saved as a ghost vehicle that shows up in later test runs. Using this feature, it could be observed that the racing car using the KIELER controller and the ghost using the *Raceyard* controller overlap at all times. Given the vast amount of simulated forces and other variables acting upon the racing car, it would be highly

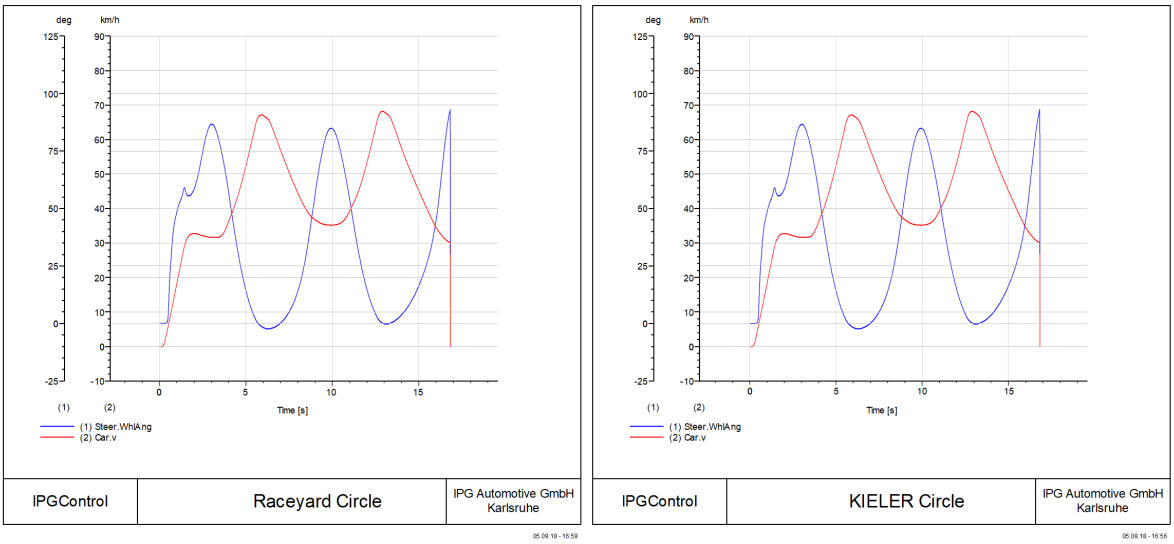


Figure 3.3: Steering angle and velocity of the Raceyard and the KIELER controller

unlikely that two controllers with different functionality would result in behaviors that are indistinguishable by eye. Therefore, given the provided exemplary data as well as the visual feedback, it is validated that the KIELER controller functions as intended.

3.3 The PI Controller in SCCharts

The preceding sections show that by using SCCharts models and integrating them into Simulink it is possible to achieve the level of functionality that is needed for the simple *Raceyard* controller. Given that the only difference in complexity between the simple and the complete *Raceyard* ECU controller other than the number of blocks used is the utilization of P and PI controllers, the questions remains whether it is possible to design a SCCharts model equivalent to a PI controller in Simulink.

A PI controller is a feedback mechanism used in control loops. A signal is measured and compared to a target value. The difference between both values, the error, is given to the PI controller, which then produces a control variable that controls a device such that the future error is minimized. The control variable in a PI Controller is the sum of two terms: The proportional term and the integral term. The proportional term is always the error value scaled by a certain factor. The integral term is the result of integrating the scaled error value over time. *Raceyard* uses P and PI controllers for its traction control system and for torque vectoring. Of special interest for remodeling in SCCharts is the I component of the controller: For each tick, the PI controller has to remember the value of its integral term and update it according to the current error value. As such, the PI controller is the only block in the *Raceyard* controller model that requires to store variables among multiple time steps.

In SCCharts internal variables can be used to store values over multiple ticks. Until

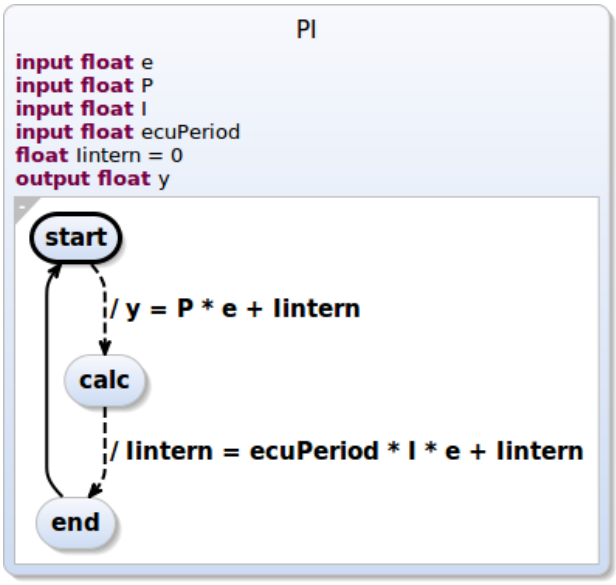


Figure 3.4: The PI Controller as realized in SCCharts

now such variables were only used to store constants or temporary results. However, as can be seen in Fig. 3.4, updating the internal variable lintern rather than overwriting it ensures the needed functionality for the integral term of the PI controller. Additionally to the error, P, I and the obligatory init inputs, the model expects one further input: the period between two calls of the ECU Controller measured in seconds. When converting the Simulink model to C code, no considerations are made about the clock rate the code operates on. Therefore, a PI controller that was simulated in Simulink using a clock rate of 1000 Hz will behave a lot slower with regards to real time when put on a micro controller that operates the controller at 50 Hz. To account for those differences in clock rates, the value added to the integral term in each tick is multiplied by the period. In this way the only difference in the behavior of the PI controller when measured in real time is due to the different sample frequencies. Note that it is possible to achieve the same result by using the product of the I factor and the period as the I input. This is how *Raceyard* accounts for the problem when using the standard PI controller block in Simulink. An additional, adequately named input, however, seemed to be the more intuitive solution, which is why it was chosen for the SCCharts model.

Fig. 3.5 shows that by integrating the SCCharts PI controller it is possible to provide the functionality of the Simulink PI controller. Given the same error, I and P inputs as well as equivalent periods both controllers will now calculate the same corrected output value. It is therefore possible in SCCharts to design a functional PI controller for use in Simulink and by extension the complete ECU controller used by *Raceyard* in the season 2018/2019.

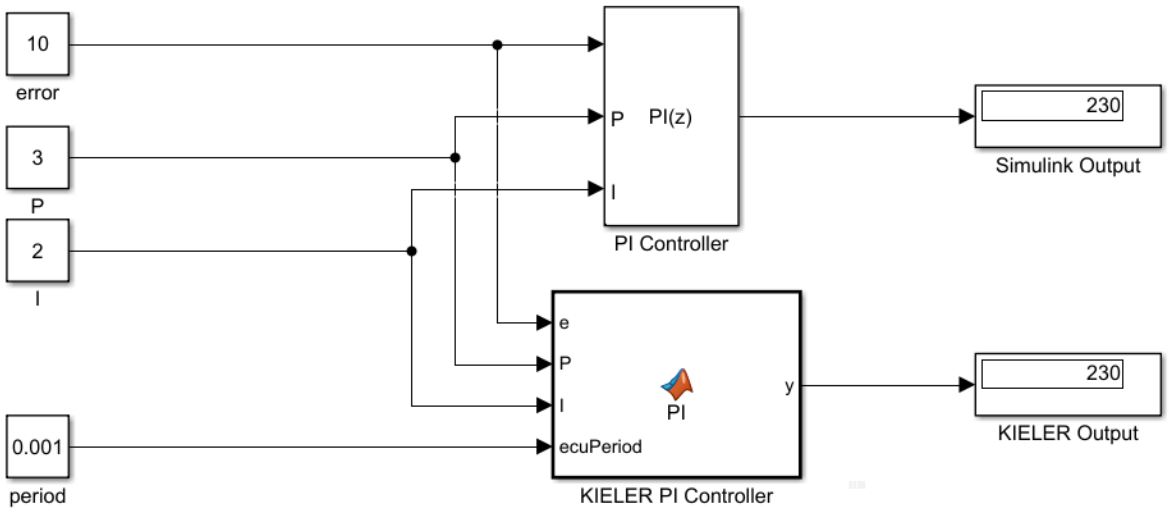


Figure 3.5: The output of the Simulink and the SCCharts PI controller after a 10 second simulation with a fixed period of 0.001 seconds

4 Performance

In order to compare the performance of both controllers, the execution time of their C code as well as their speed inside the simulation software IPG Carmaker was tested.

For comparing the C codes, both codes were put on a Raspberry Pi Version 2 and executed in batches of 1500 calls of the step() function. The time was measured immediately before and after the 1500 calls have been completed.Given the described setup, the execution time of the Simulink controller code averages at 1.648 ns, while the KIELER code averages at 6.918 ns, resulting in a slowdown of 4.2 ns when comparing the first to the latter. This is assumed to be largely due to the function overhead of the KIELER C code. Whereas the Simulink code is able to execute the calculations directly in the step() function, the KIELER code needs to call for each subsystem its wrapper(), tick() and logic()-function, resulting in a total of 15 additional function calls. The proportion of the overhead is further amplified by the fact, that only comparatively simple operations have to be executed in each subsystem. Given a situation where more sophisticated calculations had to be done the factor of 4.2 is expected to shrink. Even at its current performance, the KIELER controller would react well within the one step time limit of 50 Hz to 100 Hz set by the ECU.

While the performance of the C code is critical for usage in the real car, its performance when simulating is important as well. One typical application for the controller, where performance is of high priority, is exceeded simulation runs, when a large amount of simulation data has to be acquired in order to fine-tune the controller and for later analyzing. Here, the car usually drives on the same track many times, each time with small differences in the setup of the controller.

During simulation, IPG Carmaker provides the possibility for the user to speedup the simulation by a factor of two, three or five or by the maximum factor possible for the machine that runs the simulation. Choosing the last option, the *Raceyard* controller does usually achieve a speedup of factor 8.6 on computers used for testing the controller by the *Raceyard* team. Applying the same option on the same hardware when using the KIELER controller, a speedup of roughly 7.3 could be observed, thus resulting in a slowdown of 1.12. Though this is still noticeably slower than the original, the absolute speedup reached is well within the requirements set for testing by team *Raceyard*.

5 Related Work

Attempting to connect model based systems with Simulink or integrating part of their functionality into Simulink is a long-standing field of interest. Many of such attempts try to couple Simulink with UML in order to combine the control functionality of the former with the capability in system design of the latter. One such example is the work of Shi in 2007, where it was noted that while "Simulink provides support for embedded system design at code level [it] does not directly support their software/hardware implementation" [9]. Their attempts in behavioral mapping between Simulink and UML2 models were left with some unsolved problems, most of them having to do with Simulink's many solvers and their timing. This reflects the conclusions by Hooman in 2004, where coupling of Simulink with the UML based tool Rose RealTime was complicated due to the problem "to establish a common notion of time" [6].

Some success was achieved by Sindicos in 2011 in their work to automatically transform between Simulink models and SysML models. To this end, an open-source model-to-text generator was utilized [10], resembling parts of the workflow presented in this work.

Prior to this, in 2005, in another approach Dormido attempted to add interactivity to existing Simulink models via a 3D GUI [4]. Utilizing the software tool *Easy Java Simulations*, the inputs, outputs and the parameters of the Simulink model can be connected to Easy Java. The connection has to be done manually by the user via the Easy Java GUI. Once established, Simulink would manage the timing of the simulation by iteratively running the model, while the user could utilize the GUI to change the inputs. Of special relevance to this project is the observation by Dormido that while changing the parameters of the Simulink model is possible in Easy Java, usage of the new variable is only guaranteed by Simulink if the model was not already running. This, however, is exactly the intent by *Raceyard* to use certain global variables like maximum power in Fig. 2.5b. In this case, maximum power could be changed not often enough during runtime to warrant its own input, but for example once or twice during the endurance event, when considerations have to be made about the charging status of the accumulator. Though neither testing nor inspection of the C code indicates any reason to suspect that changes made to the global variables in question would be left unnoticed by the program, this can be due to the rather simple nature of the controller. In a model with complex internal states as they may have been present in the simulations run by Dormido, complications when changing parameters during runtime are conceivable.

In the same work Dormido also mentioned communication problems between Easy Java and Simulink that arose when the Simulink models included integrator subtypes rather than the standard block. This again could be of relevance for *Raceyard*, should more sophisticated control mechanisms be considered in the future. However, it should

be noted that Dormido encountered those complications in 2005 and many updates were released for MATLAB and Simulink since.

6 Conclusion

This work shows that KIELER and SCCharts provide the necessary functionality to design a complete controller model equivalent to the one used by *Raceyard* for its torque calculation. The identical behavior of the KIELER controller has been established during multiple tests in different environments. Furthermore, its performance has been proven to be sufficient for extensive simulation as it is e.g. necessary for lap time optimization.

Consequently, a method was presented to integrate KIELER generated C Code into MATLAB/Simulink. Design of a hybrid Simulink/SCCharts system is therefore possible. However, due to the resulting functional overhead such an approach seems expedient mostly in cases where computationally complex modules are integrated. Since the integrated KIELER model is essentially a black block in Simulink, debugging is also complicated.

Given the results in this work, a design process of the controller done solely in KIELER is conceivable. In this case, direct integration of the generated code into simulations such as IPG Carmaker, i.e. without a detour over Simulink, would be preferable. Since IPG Carmaker allows the integration of C Code, such an approach seems feasible.

Bibliography

[1] C. André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996.

[2] G. Berry. The foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454, Cambridge, MA, USA, 2000. MIT Press.

[3] G. Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, Version 3.0, Centre de Mathématiques Appliqées, Ecole des Mines de Paris and INRIA, 2004 route des Lucioles, 06902 Sophia-Antipolis CDX, France, Dec. 2002.

[4] S. Dormido, F. Esquembre, G. Farias Castro, and J. Sanchez Moreno. Adding interactivity to existing Simulink models using Easy Java Simulations. In *IEEE CONFERENCE ON DECISION AND CONTROL*, volume 44, page 4163. IEEE; 1998, 2005.

[5] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[6] J. Hooman, N. Mulyar, and L. Posta. Coupling simulink and uml models. In *Proc. Symposium FORMS/FORMATS*, pages 304–311, 2004.

[7] C. Motika. *SCCharts—Language and Interactive Incremental Implementation*. Number 2017/2 in Kiel Computer Science Series. Department of Computer Science, 2017. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel.

[8] C. Motika, S. Smyth, and R. von Hanxleden. Compiling SCCharts—A case-study on interactive model-based compilation. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*, volume 8802 of *LNCS*, pages 461–480, Corfu, Greece, Oct. 2014.

[9] J. Shi. *Model and tool integration in high level design of embedded systems*. PhD thesis, KTH, 2007.

[10] A. Sindico, M. Di Natale, and G. Panci. Integrating sysml with simulink using open-source model transformations. In *SIMULTECH*, pages 45–56, 2011.

[11] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 372–383, Edinburgh, UK, June 2014. ACM.

[12] N. Wechselberg, A. Schulz-Rosengarten, S. Smyth, and R. von Hanxleden. Augmenting state models with data flow. In M. Lohstroh, P. Derler, and M. Sirjani, editors, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS, pages 504–523. Springer International Publishing, 2018.